



新浪微博：C语言图书
腾讯QQ：4006751066



C语言学习路线图

C语言必须知道的 300个问题

快速服务：微博、QQ在线服务

自学视频：40集大型多媒体自学视频

海量资源：模块库、案例库、素材库、题库



明日科技 编著

本书提供了内容丰富的配套资源，可以登录www.tup.com.cn，找到本书后，在该页面的“网络资源”超链接处下载。也可以访问本书的新浪微博，根据提示链接下载。



清华大学出版社

C语言学习路线图

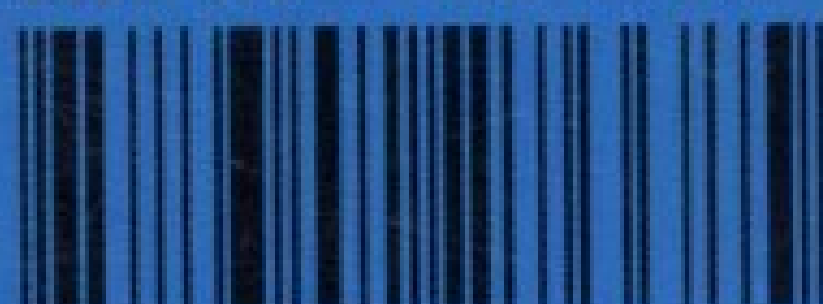
C语言的学习整体解决方案

C语言学习路线图，为读者朋友提供了从入门到实际项目开发所需要的各方面必备知识，提供了较为完善的学习整体解决方案，搭起了从学校走向社会的桥梁。各个品种既有前后关联，也可以独立使用。从而避免了像以前那样，学完一本书之后，仍然无所适从，既不会做项目也不知道接下来该学什么，以至于半途而废的困惑。

C语言开发入门及项目实战

定价：59.80元

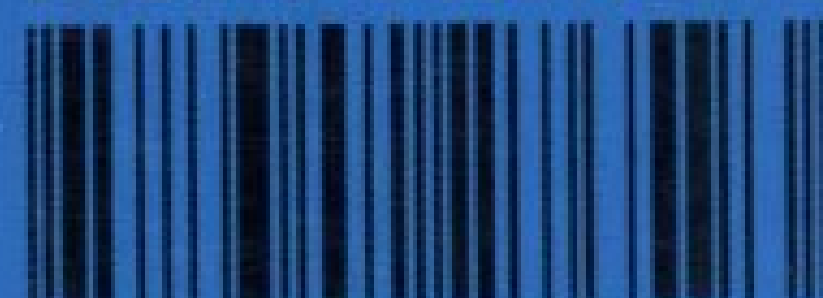
ISBN 978-7-302-27668-5



C语言经典编程282例

定价：49.80元

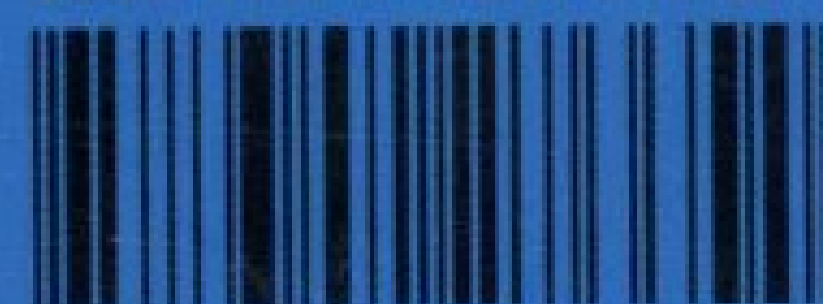
ISBN 978-7-302-27659-3



C语言常用算法分析

定价：39.80元

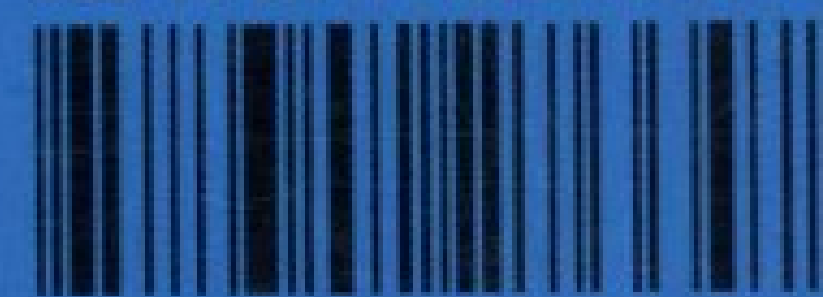
ISBN 978-7-302-27665-4



C语言函数参考手册

定价：49.80元

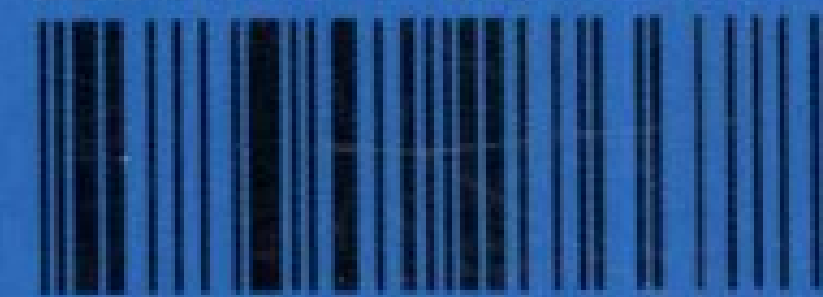
ISBN 978-7-302-27664-7



C语言必须知道的300个问题

定价：49.80元

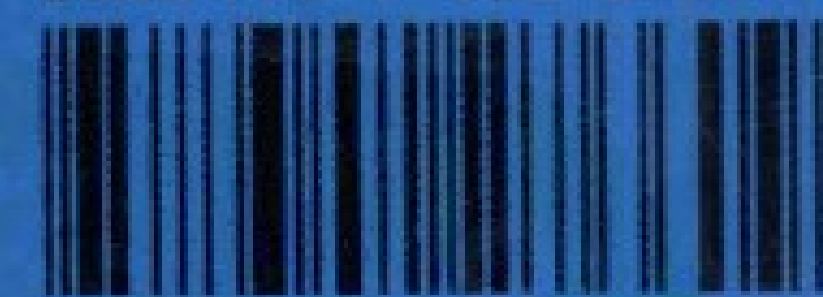
ISBN 978-7-302-27667-8



C语言项目案例分析

定价：49.80元

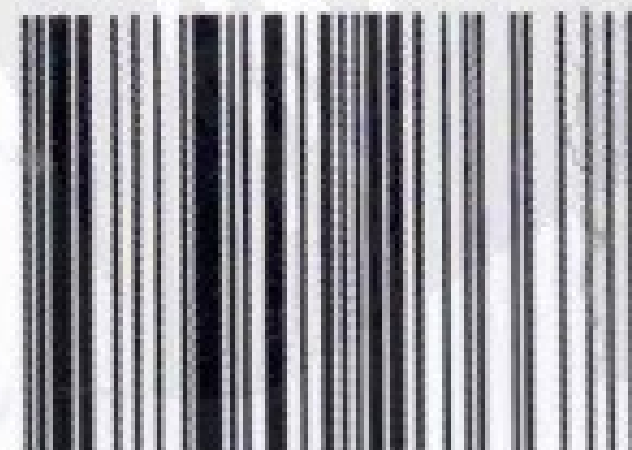
ISBN 978-7-302-27660-9



清华大学出版社数字出版网站

WQBook 
www.wqbook.com

ISBN 978-7-302-27667-8



9 787302 276678 >

定价：49.80元

C 语言学习路线图

C 语言必须知道的 300 个问题

明日科技 编著

（模块库、案例库、素材库、题库）

（微博、QQ、论坛技术支持）

清华大学出版社

北 京

前言

Preface



学会站在巨人的肩膀上！

程序员在求职时，经常会被问到有关开发经验的问题。例如，“从事了多少年的开发？”、“参与了哪些大型项目？”……为何面试官对项目经验丰富的求职者青睐有加？究其根源，是由于程序开发是一门实践性非常强的学科。正所谓“纸上得来终觉浅，绝知此事要躬行”。编程经验是程序开发者在长期的实践中逐步积累的宝贵的技术资源库，其中包含各种习惯用法、知识理论、代码片段和应用技巧等。要想成为经验丰富的编程高手，除了勤学苦练，也要学会站在前人的肩膀上，通过分析别人的代码而有所领悟，或者借鉴他人的经验技巧形成自己的技能，在认识错误与解决问题中不断进步。这也是每个编程者的必由之路。

本书汇集了 300 个一线开发人员常见的疑难问题，每个问题都给出了详细的解答与点评，图文并茂、难易并举，通过对本书的学习，读者可以尽享前人的开发经验，迅速提升个人的开发能力。

本书内容

本书以基础知识为框架，介绍了 C 语言各部分知识所对应的常见开发疑难问题，并作了透彻地解析。全书共分为 17 章，主要内容包括初识 C 语言，一个简单的 C 程序，算法入门，常用数据类型，运算符与表达式，输入/输出函数，选择、分支结构程序设计，循环结构，数组，函数编程基础，指针解析，常用数据结构，位运算操作符，存储管理，预处理和函数类型，文件的读写操作和图形图像处理。

为了更清晰地阐述问题并给出详尽的解决方案，本书设置了以下栏目。

☒ 问题阐述

对问题进行详细阐述，对复杂问题给出问题产生的条件，并对问题中需要解决的部分作出说明。

☒ 专家解答

根据问题进行具体分析，给出相应的解题思路及正确答案，并提供该问题涉及的技术知识。

☒ 专家点评

对问题及解答进行总结，为以后的问题解决提供思路；此外，还会列出一些有关此问题应该注意的事项，以及对该问题的一些拓展。



本书特色

☒ 贴近应用

本书精选的疑难问题都是在实际项目开发中经常会遇到的，主要目的就是为读者答疑解惑。

☒ 横向链接

本书知识框架与《C 语言开发入门及项目实战》一书相对应，可以在使用《C 语言开发入门及项目实战》一书进行基础学习之后，使用本书丰富并提高技能。

☒ 解析透彻

本书对每个问题的相关知识都作了细致地讲解，并进行知识拓展，使读者不仅知其然而且知其所以然。

☒ 授人以渔

本书在讲解技术的同时，还注重对读者能力的培养，使读者真正掌握分析问题与解决问题的能力。

本书配套资源

本书提供了内容丰富的配套资源，包括源程序、素材，以及模块库、案例库、题库、素材库等多项辅助内容，读者可以通过如下方式获取。

第 1 种方式：

(1) 登录 www.tup.com.cn，在网页右上角的搜索文本框中输入本书书名（注意区分大小写和留出空格），或者输入本书关键字，或者输入本书 ISBN 号（注意去掉 ISBN 号间隔线“-”），单击“搜索”按钮。

(2) 找到本书后单击超链接，在该书的网页下侧单击“网络资源”超链接，即可下载。

第 2 种方式：

访问本书的新浪微博：C 语言图书，找到配套资源的链接地址进行下载。

读者人群

本书非常适合以下人员阅读：

- ☒ 从事 C 语言编程的开发人员
- ☒ 有一定语言基础，想进一步提高技能的人员
- ☒ 大中专院校的老师 and 学生
- ☒ 即将走上相关工作岗位的大学毕业生
- ☒ 相关培训机构的老师和学员
- ☒ C 语言编程爱好者



Note



读者服务&本书勘误

读者在使用本书过程中遇到的所有问题，均可通过以下方式联系我们。

1. 新浪微博：C 语言图书。

及时发布读者答疑、本书勘误、配套资料更新等内容。

2. 腾讯 QQ 群：4006751066

3. 登录网站：www.mingribook.com，在论坛、勘误发布、读者纠错、技术支持、读者之家等栏目中的相关模块中提问、留言或查看。

本书作者

本书由明日科技组织编写，参与编写的有孙秀梅、曹飞飞、王雪、朱晓、赵永发、李鑫、潘凯华、刘欣、李慧、高春艳、王小科、赵会东、李继业、赛奎春、王国辉、陈丹丹、李伟、杨丽、李丽、刘龄龄、王明招、孙茜、陈英、肖鑫等。

由于作者水平有限，疏漏和不足之处在所难免，敬请广大读者朋友批评指正。

编 者

目 录

Contents



第 1 章 初识 C 语言..... 1

- 问题 1 C 语言是如何发展起来的? 2
- 问题 2 什么是 ANSI 标准? 2
- 问题 3 C 语言编写程序的优点有哪些? ... 3
- 问题 4 如何规避 C 语言的不足之处? 4
- 问题 5 C 语言的应用领域有哪些? 5
- 问题 6 什么是 C99 标准? 与 C89 标准
相比, C99 标准有哪些
新特性? 6
- 问题 7 C 语言是 C++ 的子集吗? 10
- 问题 8 C 语言程序的开发过程是
怎样的? 11
- 问题 9 什么是编译程序和解释程序? 12
- 问题 10 ANSI C 的编译限制有哪些? 13

第 2 章 一个简单的 C 程序..... 14

- 问题 11 C 语言的入口函数是什么? 15
- 问题 12 C 语言程序由哪些部分组成? 16
- 问题 13 如何在 Turbo C 2.0 中输入一个
程序? 16
- 问题 14 如何在 Visual C++ 6.0 中运行
一个 C 程序? 18
- 问题 15 如何在 Visual C++ 2008 中运行
一个 C 程序? 22
- 问题 16 如何提高程序的可读性? 26
- 问题 17 什么是关键字? C 语言的关键
字有哪些? 27
- 问题 18 什么是标识符? 使用标识符的
注意事项是什么? 29
- 问题 19 void 关键字都有哪些用途? 30
- 问题 20 什么是匈牙利命名约定? 它是否

是好的约定? 31

第 3 章 算法入门..... 33

- 问题 21 为什么说算法是程序设计的
灵魂? 34
- 问题 22 算法的特性有哪些? 34
- 问题 23 如何评价一个算法的好坏? 35
- 问题 24 算法的表示方法都有哪些? 36
- 问题 25 算法的基本结构是什么? 40
- 问题 26 算法有哪几类? 42
- 问题 27 算法的效率度量方法有哪些? .. 42
- 问题 28 什么是算法的时间复杂度? 43
- 问题 29 什么是算法的空间复杂度? 44
- 问题 30 什么是分治法算法思想? 45

第 4 章 常用数据类型..... 46

- 问题 31 声明变量和定义变量的区别
是什么? 47
- 问题 32 在开发时如何决定使用哪种
数据类型? 48
- 问题 33 什么是常量? 如何区分常量
和变量? 48
- 问题 34 各种数据类型所占的内存是
多少? 49
- 问题 35 字符与字符串的差别有
哪些? 50
- 问题 36 变量是否必须初始化? 51
- 问题 37 为什么会发生数据溢出? 如何
避免数据溢出? 52
- 问题 38 局部变量和全局变量能否
重名? 53



问题 39	全局变量可不可以定义在可被多个.C 文件包含的头文件中? 为什么?	53
问题 40	如何引用一个已经定义过的外部变量?	54
问题 41	全局变量和局部变量的存储方式有什么区别?	55
问题 42	整型常量的存储形式是怎样的?	55
问题 43	整型常量的表示形式有哪几种?	56
问题 44	使用了没定义的变量会有什么现象?	56
问题 45	static 关键字有什么作用?	57
问题 46	const 关键字有什么作用?	59
问题 47	const 与#define 相比有何优点?	60
问题 48	sizeof 不是函数吗?	61
问题 49	float 类型数如何与 0 值比较?	63
问题 50	静态变量与自动变量的区别有哪些?	64

第 5 章 运算符与表达式 66

问题 51	运算符的优先级和结合性是怎样的?	67
问题 52	如何区分 “,” 是运算符还是分隔符?	68
问题 53	C 语言如何解释 $x=a+=b+c$?	69
问题 54	$x=x+1$, $x+=1$, $x++$, 哪个效率最高?	70
问题 55	什么是运算符的目? 怎样进行区分?	70
问题 56	使用 “++” 和 “--” 运算符需要注意些什么?	71
问题 57	如何理解 $i+++j$?	71
问题 58	赋值表达式中什么是左值和右值? 数组名作为左右值时又具有怎样的意义?	72
问题 59	如何确定条件表达式的结果的	

数据类型? 73

问题 60	“%” 运算符是否可以对小数进行运算?	74
问题 61	“/” 运算符得到的结果一定为整数吗?	75
问题 62	在进行多种数据类型混合运算的时候, 数据类型自动转换有哪些规则?	76
问题 63	C 语言中有哪些简化的运算表达式?	77
问题 64	使用逻辑表达式需要注意哪几点问题?	77
问题 65	$i++*i++$ 这样的表达式为什么不能得到预期的结果?	78
问题 66	$a[i]=i++$; 这样的代码正确吗? ...	79
问题 67	编写表达式时需要注意什么?	79
问题 68	如何理解 $c=a,b$?	80
问题 69	为无符号类型变量赋值时, 数据类型应怎样转换?	81
问题 70	C 语言表达式的求值顺序总是按照运算符的结合性保证“自左至右”或者“自右至左”吗?	81

第 6 章 输入/输出函数 83

问题 71	函数 printf() 的基本格式是什么?	84
问题 72	如何认识 printf() 函数的格式字符?	85
问题 73	函数 printf() 的标志有几种? 如何使用?	91
问题 74	如何控制输出最小宽度?	91
问题 75	如何控制输出精度?	92
问题 76	如何控制输出长度?	93
问题 77	如何动态设置输出宽度和精度?	94
问题 78	printf() 函数的返回值是什么?	95
问题 79	如何理解输出列表?	96



问题 80	函数 scanf()的基本格式是什么?	96	问题 104	if 语句与 switch 语句的优缺点是什么?	127
问题 81	scanf()函数的格式字符是什么?	97	问题 105	switch 语句中的 default 关键字是否必须?	128
问题 82	使用 scanf()函数应注意的问题是什么?	100	问题 106	break 关键字在 switch 语句中应注意什么?	128
问题 83	scanf()函数的返回值是什么?	101	问题 107	如何正确判断 if 和 else 的匹配?	130
问题 84	如何使用 getchar()函数?	102	问题 108	switch 和 case 后的表达式值的类型是否可以是浮点型?	132
问题 85	getch()函数如何使用?	104	问题 109	区段划分型条件有什么技巧?	133
问题 86	如何应用 gets()函数?	104	问题 110	default 必须写在所有 case 之后吗?	134
问题 87	如何应用 putchar()函数?	105	第 8 章	循环结构	136
问题 88	puts()函数该如何应用?	106	问题 111	循环结构的基本概念是什么?	137
问题 89	如何控制多数值的输入?	107	问题 112	while 语句的基本格式是什么?	138
问题 90	如何输入字符数组?	108	问题 113	while 循环应注意什么问题?	139
第 7 章	选择、分支结构程序设计	110	问题 114	for 循环语句的基本格式是什么?	140
问题 91	5>4>3 为什么不成立——谈谈关系表达式的值	111	问题 115	for 语句的三个表达式都是必须的吗?	142
问题 92	=和==如何区分?	112	问题 116	do...while 语句的基本格式是什么?	144
问题 93	什么叫逻辑短路?	113	问题 117	分号在循环体中的作用?	146
问题 94	if 语句的基本形式有哪些? 如何应用?	114	问题 118	while 与 do...while 的区别?	148
问题 95	浮点数的相等比较是否可以用==?	116	问题 119	什么是循环嵌套?	149
问题 96	关系运算符和数学不等号有什么区别?	117	问题 120	循环嵌套的结构是怎样的?	151
问题 97	if 语句后面一定不能写分号吗?	118	问题 121	如何正确使用循环嵌套?	153
问题 98	这个程序为什么多执行了好多语句?	119	问题 122	死循环是怎样产生的?	154
问题 99	不用关系表达式和逻辑表达式做条件	120	问题 123	怎样提高循环语句的效率?	156
问题 100	怎样理解复合语句中的变量?	122	问题 124	continue 语句的基本作用是什么?	158
问题 101	如何进行 if 语句的嵌套?	123			
问题 102	条件运算符“?:”怎样应用?	124			
问题 103	switch 语句的基本格式是什么?	125			



- 问题 125 `break` 语句的基本作用是什么? 159
- 问题 126 `goto` 语句的基本格式是什么? 如何使用? 161
- 问题 127 `goto` 语句的缺陷是什么? 163
- 问题 128 如何选择循环语句? 165
- 问题 129 如何判定循环结束和提前结束? 165
- 问题 130 如何避免循环中的初值错误问题? 167

第 9 章 数组 170

- 问题 131 什么是数组? 其存储有何特点? 171
- 问题 132 数组的维数该如何理解? 171
- 问题 133 一维数组是怎样定义的? 172
- 问题 134 如何引用一维数组元素? 173
- 问题 135 如何初始化一维数组? 174
- 问题 136 如何设计数组的排序算法? 175
- 问题 137 如何定义二维数组? 176
- 问题 138 如何引用二维数组元素? 178
- 问题 139 如何初始化二维数组? 179
- 问题 140 如何定义字符数组? 180
- 问题 141 如何初始化字符数组? 180
- 问题 142 如何引用字符数组? 181
- 问题 143 如何进行字符数组的复制? 182
- 问题 144 如何进行字符数组的连接? 183
- 问题 145 如何进行字符串的比较? 185
- 问题 146 如何测定字符串的长度? 187
- 问题 147 如何进行字符串大小写的相互转换? 188
- 问题 148 如何计算字符串中有多少个单词? 190
- 问题 149 `gets()` 函数和 `scanf()` 函数在输入字符串时有何区别? 191
- 问题 150 `puts()` 函数和 `printf()` 函数在输出字符串时有何区别? 192

- 问题 151 数组与指针的区别是什么? .. 193
- 问题 152 为什么作为函数形参的数组和指针可以互换? 195
- 问题 153 为什么数组名作参数传递给子函数时, 子函数可以改变主函数中数组的值? 196
- 问题 154 C 语言中有动态数组吗? 197
- 问题 155 如何实现动态二维数组? 199
- 问题 156 `strcpy()` 函数可以复制字符串的一部分吗? 201
- 问题 157 字符串和字符数组有什么区别? 202
- 问题 158 ‘\0’和“\0”有什么区别? 203
- 问题 159 字符数组占用内存怎样算? 204
- 问题 160 用字符数组和指针两种方式定义的字符串有什么不同? 204

第 10 章 函数编程基础 206

- 问题 161 什么是函数? 如何分类? 207
- 问题 162 如何定义无参函数? 209
- 问题 163 如何定义有参函数? 210
- 问题 164 什么是空函数? 作用是什么? 210
- 问题 165 什么是形参和实参? 如何使用? 211
- 问题 166 如何从函数返回? 213
- 问题 167 函数返回值你了解多少? 214
- 问题 168 如何进行函数的一般调用? 215
- 问题 169 函数调用的基本方式有几种? 各是什么? 216
- 问题 170 函数调用应具备哪些条件? 216
- 问题 171 如何进行函数的嵌套调用? 218
- 问题 172 什么是递归调用? 如何实现? 219
- 问题 173 函数如何将数组元素作为



实参?	222
问题 174 如何将数组名作为函数参数?	224
问题 175 如何将多维数组名作为函数参数?	227
问题 176 什么是局部变量?	228
问题 177 什么是全局变量? 如何应用?	229
问题 178 存储方式有哪几种? 分别是什么?	232
问题 179 如何使用 auto 关键字?	233
问题 180 什么是静态变量? 如何实现?	236
问题 181 什么是寄存器变量? 如何实现?	238
问题 182 如何声明外部变量?	239
问题 183 如何调用编译后的函数?	240
问题 184 如何限定外部变量的使用范围?	241
问题 185 如何使用函数调用实现对字符串的统计?	242
问题 186 main()函数有什么作用?	243
问题 187 什么是内部函数?	243
问题 188 什么是外部函数? 怎么用?	244
问题 189 static()函数与普通函数有什么区别?	245
问题 190 形参和实参有什么区别?	246
第 11 章 指针解析	248
问题 191 什么是指针? 什么是指针变量?	249
问题 192 如何创建指针?	250
问题 193 如何初始化指针?	251
问题 194 如何使用指针?	252
问题 195 函数中如何传递指针?	254
问题 196 指针、数组和地址之间的关系是什么?	255
问题 197 如何进行指针运算?	256
问题 198 如何使用指针操作数组?	258
问题 199 如何用指针表示多维数组?	260
问题 200 如何使用指针操作多维数组?	261
问题 201 如何用指针为函数传递数组?	263
问题 202 如何用指针表示字符串?	264
问题 203 如何使用字符串指针作为函数参数?	265
问题 204 字符数组和字符指针的区别是什么?	266
问题 205 什么是指针数组?	267
问题 206 如何使用指针数组处理字符串?	268
问题 207 如何将指针数组作为函数的参数?	269
问题 208 什么是指向指针的指针?	270
问题 209 二级指针如何应用于一维数组?	271
问题 210 如何实现二级指针对二维数组的操作?	272
问题 211 二级指针如何操作字符串数组(指针数组)?	273
问题 212 如何理解返回指针的函数?	275
问题 213 什么是指向函数的指针?	277
问题 214 如何用 const 控制指针?	278
问题 215 什么是“野指针”?	279
问题 216 main()函数的指针数组形参是怎么回事?	279
问题 217 void 指针就是空指针吗? 它有什么作用?	281
问题 218 指针是一种特殊的变量, 只能用来保存地址。这句话对吗?	282
问题 219 字符指针、浮点数指针以及函数指针这三种类型的变量哪个占用的内存最大? 为什么?	282



Note

- 问题 220 一个 32 位的机器, 该机器的
指针是多少位? 283

第 12 章 常用数据结构 284

- 问题 221 空结构体所占的内存是
多少? 285
- 问题 222 在 C 语言中, 一个结构体可以
包含指向自己的指针吗? 286
- 问题 223 struct person {...}; person a;
为什么编译出错? 287
- 问题 224 怎样从/向数据文件读/写
结构? 289
- 问题 225 枚举与#define 宏的区别有
哪些? 290
- 问题 226 如何看待枚举类型, 枚举类型
的优点是什么? 291
- 问题 227 关键字 typedef 的功能是
什么? 292
- 问题 228 类型定义是否允许嵌套? 294
- 问题 229 typedef 与#define 宏的相似之处
与不同之处是什么? 295
- 问题 230 什么是散列法? 296
- 问题 231 大小端模式对 union 类型数据
有什么影响? 297
- 问题 232 如何为联合体变量赋
初值? 298
- 问题 233 如何证明联合体变量的所有成员
是共享一个内存单元的? 300
- 问题 234 堆和栈的区别是什么? 301
- 问题 235 举例说明, 什么是静态链表?
什么是动态链表? 302
- 问题 236 单向链表、双向链表和循环链
表有什么区别? 304
- 问题 237 如何在链表中的指定位置插入
结点? 305
- 问题 238 如何删除链表中指定位置的
结点? 306
- 问题 239 如何创建一个动态链表? 308
- 问题 240 指向结构体数组的指针如何
应用? 310

第 13 章 位运算操作符 312

- 问题 241 什么是位运算? 位运算符包括
哪些? 313
- 问题 242 移位运算中如何补位? 314
- 问题 243 移位运算符与加减运算符的
优先级哪个较高? 315
- 问题 244 什么是循环移位? 316
- 问题 245 什么是位段? 其优点是
什么? 317
- 问题 246 如何正确使用位段? 318
- 问题 247 数据在计算机中的存储单位有
哪些? 有几种存储形式? 320

第 14 章 存储管理 322

- 问题 248 与内存息息相关的重要概念
有哪些? 323
- 问题 249 指针指向不合法引起了哪些
内存问题? 324
- 问题 250 内存分配与释放引起的常见
问题有哪些? 325
- 问题 251 什么是内存越界? 什么是内存
泄露? 二者是如何产生的? ... 326
- 问题 252 C 语言提供了哪些动态内存
分配函数? 327
- 问题 253 malloc()函数与 calloc()函数有
什么区别? 328
- 问题 254 内存耗尽怎么办? 328
- 问题 255 动态内存会被自动释
放吗? 330
- 问题 256 高位优先与低位优先的不同
之处是什么? 330
- 问题 257 free()和 delete()怎样处理
指针? 331
- 问题 258 怎样利用好敏感的内存
资源? 333

第 15 章 预处理和函数类型 335

- 问题 259 在头文件中#if、_STDC_等字符
起什么作用? 336
- 问题 260 如何书写多条语句宏? 337



- 问题 261 预处理中#和##运算符是什么意思? 338
- 问题 262 一个头文件可以包含另一个头文件吗? 339
- 问题 263 #include<>和#include“”有什么区别? 340
- 问题 264 什么是无参宏定义? 341
- 问题 265 什么是带参宏定义? 342
- 问题 266 怎样写参数个数可变的宏? 343
- 问题 267 #pragma 预处理的作用是什么? 345
- 问题 268 条件编译的表达形式有哪些? 346
- 问题 269 如何应用内部函数? 347
- 问题 270 如何应用外部函数? 348

第 16 章 文件的读写操作 351

- 问题 271 各个读写操作的区别是什么? 352
- 问题 272 C 语言文件有哪几类? 354
- 问题 273 怎样写数据文件,使之可以在不同字大小、字节顺序或浮点格式的机器上读入? 355
- 问题 274 能否使用 fflush()函数清除多余的输入? 356
- 问题 275 fopen()函数打开文件失败的原因是什么? 357
- 问题 276 为什么打开文件后要及时关闭? 358
- 问题 277 文件的打开方式有哪些? 358
- 问题 278 如何正确使用 putchar()函数和 getchar()函数? 360
- 问题 279 getchar()函数、getch()函数和 getche()函数的区别是什么? 361
- 问题 280 使用 printf()函数和 scanf()函数需要注意什么? 362
- 问题 281 printf()函数有哪些参数? 363

- 问题 282 scanf()函数的格式控制包括哪些? 364
- 问题 283 printf()函数和 scanf()函数格式符的修饰符“*”有什么作用? 366
- 问题 284 fscanf()函数、fprintf()函数与 scanf()函数和 printf()函数有什么不同? 367
- 问题 285 如何判断文件的结束? 368

第 17 章 图形图像处理 371

- 问题 286 为什么在使用图形函数时要首先初始化图形模式? 372
- 问题 287 怎样初始化图形模式? 372
- 问题 288 初始化时提示“BGI Error: Graphics not initialized (use 'initgraph')”怎么办? 374
- 问题 289 怎样利用 C 语言建立独立的图形运行程序? 375
- 问题 290 TC 中有几个画线函数? 如何使用? 376
- 问题 291 TC 中有几个画矩形函数? 如何使用? 377
- 问题 292 TC 中有几个画圆函数? 如何使用? 379
- 问题 293 如何使用 C 语言填充封闭图形? 380
- 问题 294 TC 中有几个和光标有关的函数? 怎样使用? 382
- 问题 295 如何在图形模式下输出文本? 383
- 问题 296 背景色、线条颜色和填充颜色有什么区别? 何时使用? 386
- 问题 297 怎样记住那么多的颜色? 387
- 问题 298 线条样式和填充样式都有哪些? 怎样设置? 388
- 问题 299 怎样复制图形? 392
- 问题 300 怎样在 C 语言中制作动画? 394

第1章

初识 C 语言

- ▶▶ C 语言是如何发展起来的?
- ▶▶ 什么是 ANSI 标准?
- ▶▶ C 语言编写程序的优点有哪些?
- ▶▶ 如何规避 C 语言的不足之处?
- ▶▶ C 语言的应用领域有哪些?
- ▶▶ 什么是 C99 标准? 与 C89 标准相比, C99 标准有哪些新特性?
- ▶▶ C 语言是 C++ 的子集吗?
- ▶▶ C 语言的开发过程是怎样的?
- ▶▶ 什么是编译程序和解释程序?
- ▶▶ ANSI C 的编译限制有哪些?



问题阐述

Note

C 语言产生、发展至今都经历了哪些阶段？是如何发展成熟的？

专家解答

在学习 C 语言之前，要先了解一下 C 语言的发展。

最初的操作系统等软件主要是使用汇编语言进行编写，由于汇编语言有依赖于硬件、程序可读性和移植性都比较差等特点，而高级语言又难以实现直接对硬件等进行操作，故需发展一种新的语言，使其既具有高级语言的特性，又具有低级语言的特性。于是，C 语言应运而生。

1970 年，UNIX 的开山鼻祖——美国贝尔实验室的 Ken Thompson 设计出了既简单又很接近硬件的 B 语言（取 BCPL 的第一个字母），并用 B 语言编写了第一个 UNIX 操作系统。

1972 年，Dennis Ritchie 在 B 语言的基础上设计出了 C 语言（取 BCPL 的第二个字母）。C 语言既保持了 BCPL 和 B 语言的优点（精练、接近硬件），又克服了它们的缺点（过于简单、数据无类型等）。最初的 C 语言只是为描述和实现 UNIX 操作系统提供一种工作语言而设计的。

1975 年，UNIX 第 6 版公布后，C 语言的突出优点引起了人们的普通关注。1977 年出现了不依赖于具体机器的 C 语言编译文本《可移植 C 语言编译程序》，简化了 C 移植到其他机器时所做的工作。

1978 年，Brian W.Kernighian 和 Dennis M.Ritchie 共同出版了影响深远的《The C Programming Language》一书。

最初，C 语言运行于 AT&T 的多用户、多任务的 UNIX 操作系统上。后来，Ritchie 用 C 语言改写了 UNIX C 的编译程序，UNIX 操作系统的开发者 Ken Thompson 又用 C 语言成功地改写了 UNIX，从此开创了编程史上的新篇章。由此，UNIX 成为第一个不是用汇编语言编写的主流操作系统。

专家点评

C 语言在当前仍是比较流行的计算机高级语言，被广泛应用于各领域。C 语言也是计算机编程的基础语言学科，所以大家应该学好 C 语言，为编程打下坚实的基础。

问题 2 什么是 ANSI 标准？

问题阐述

在使用 C 语言进行编程时，最常被提及的就是 ANSI C 标准，那么什么是 ANSI C 标



准？它是怎样被建立起来的？

专家解答

1983年，美国国家标准协会（ANSI）委任一个委员会 X3J11 对 C 语言进行标准化并于当年颁布了第一个 C 语言草案（83ANSI C），后又于 1987 年颁布了另一个 C 语言标准草案（87ANSI C）。经过漫长而艰苦的工作，该委员会的研究成果于 1989 年 12 月 14 日被正式批准为 ANSI X 3.159—1989 并于 1990 年春天颁布。ANSI C 主要标准化了现存的实践，同时增加了一些来自 C++ 的内容（主要是函数原型）并支持多国字符集（包括备受争议的三字符序列）。此外，ANSI C 标准还规定了 C 运行期库例程的标准。

一年后，该标准被接受为国际标准——ISO/IEC 9899:1990 并被广泛采用。在美国国内，该标准（在这里它被称做 ANSI/ISO 9899—1990 [1992]）也取代了早先的 X3.159。作为一个 ISO 标准，它会以发行技术勘误和标准附录的形式不断更新。

1994 年，技术勘误 1（TC1）对标准中 40 个地方作了修正，多数都是小的修改或明确；而标准附录 1（NA1）增加了大约 50 页的新材料，多数是规定国际化支持的新库函数。1995 年，TC2 增加了更多的小修改。

1999 年，ANSI 标准的一个重大修订——C99 颁布，并于 2000 年 3 月被 ANSI 采用。

ANSI 标准的数个版本，包括 C99 和原始的 ANSI 标准，都涉及一个“基本原理”（Rational），解释它的许多决定并讨论了很多细节问题，包括本文中提及的某些内容。不过由于未得到主流编译器厂家的支持，C99 并未被广泛采用。

专家点评

目前流行的 C 编译器一般都是以 ANSI 标准为基础的，各种版本的 C 语言编译器系统虽然基本部分是相同的，但是也有一些不同，读者在使用的时候要注意识别。

问题 3 C 语言编写程序的优点有哪些？

问题阐述

C 语言能够存在并发展至今，其生命力之强可见一斑。这其中一定是有着某些不可替代的优点，那么 C 语言编写程序的优点都有哪些呢？

专家解答

为了方便读者理解，下面对 C 语言的每条特点进行详细的解说。

（1）程序结构简洁、紧凑、规整，表达式简练、使用灵活。

（2）编写的程序可读性强，编译效率高。

（3）具有丰富的运算符，多达 34 种。丰富的数据类型与丰富的运算符相结合，使 C 语言具有表达灵活和效率高等特点。



Note



Note

(4) 数据类型种类繁多。C 语言具有 5 种基本的数据类型和多种构造数据类型以及复合的导出类型，同时还提供了与地址密切相关的指针机器运算符。指针可以指向各种类型的简单变量、数组、结构和联合，乃至函数等。此外，C 语言还允许用户自己定义数据类型。

(5) 是一种结构化程序设计语言，特别适合大型程序的模块化设计。C 语言具有编写结构化程序所必需的基本流程控制语句，C 语言程序是由函数集合构成的，函数各自独立，并且作为模块化设计的基本单位。

说明：

C 语言的源文件，可以分割成多个源程序，分别进行编译，然后连接起来构成可知性的目标文件，为开发大型软件提供了极大的方便。C 语言还提供了多种存储属性，使数据可以按其需要在相应的作用域起作用，从而提高了程序的可靠性。

(6) 语法限制不太严格，程序设计自由度大。例如，对数组下标越界不作检查，由程序编写者自己保证程序的正确。一般的高级语言语法检查比较严，能检测出几乎所有的语法错误，而 C 语言允许程序编写者有较大的自由度，因此放宽了语法的检查。程序员应当仔细检查程序，保证其正确，而不要过分依赖 C 语言编译程序去查错。

(7) 允许直接访问物理地址，能进行位 (bit) 操作，能实现汇编语言的大部分功能，可以直接对硬件进行操作。因此，C 语言既具有高级语言的功能，又兼容低级语言的许多功能，可用来编写系统软件。

(8) 生成的目标代码质量高，程序执行效率高。它一般只比汇编程序生成的目标代码率低 10%~20%。

(9) 具有较高的可移植性。它的语句基本上无须修改就能用于各种型号的计算机和各种操作系统。

C 语言是处于汇编语言和高级语言之间的一种中间型程序设计语言，常被称为中级语言。它既有高级语言的基本特点，又具有汇编语言面向硬件和系统，可以直接访问硬件的功能。

专家点评

C 语言的这些优点，读者仅通过这里的介绍还不能深刻理解和体会，待对 C 语言有了一定的了解之后再回顾一下，就会体会到这些优点了。但由于 C 语言的限制少、灵活性大、功能强，所以对程序员有较高的要求。在使用 C 语言进行编程时，需要有足够的细心和耐心。

问题 4 如何规避 C 语言的不足之处？

问题阐述

C 语言虽然具有功能强、灵活性大、可移植性强、应用广泛等优点，但是同时也存在



着一定的缺点，导致在学习和编程时造成一定的困难和错误。那么在编程时应该如何规避C语言的不足之处呢？

专家解答

C语言是一门非常流行的编程语言，被许多平台选用，具有许多的优点。然而，正如人们常说的“事物都具有矛盾性”，有利必有弊，在看到其种种优点之时，千万不要忽视其存在的不足和缺陷。下面的分析不是为了打击大家学习C语言的积极性，而是为了使大家更好地了解和学习C语言。

(1) C语言语法限制不太严格，程序设计自由度大。“限制”与“灵活”是对立的，强调“灵活”就会放松“限制”。这就对使用C语言进行编程提出了更高的要求，程序员要对程序设计更熟悉。例如，C语言对数组下标越界不检查，容易造成数据在内存中的混乱。

(2) C语言具有丰富的运算符，多达34种。丰富的数据类型与丰富的运算符相结合，使C语言具有表达灵活和效率高等特点。然而，这却增加了使用C语言的难度，这些运算符分为众多优先级，不容易记忆，可能混淆而产生错误。

(3) C语言存在着一些不应该存在的语法限制。例如，switch语句由case结构组成，每个case结束之后都要使用一个break来跳出case结构，否则将会继续执行下面的case，这样就会造成错误。

(4) C语言中许多运算符被“重载”，具有不同的意义，甚至有些关键字也具有好几种意义。例如，void作为函数的返回类型，表示不返回任何值；在指针声明中，表示通用指针类型；在参数列表中，表示没有参数。

专家点评

尽管C语言存在着诸多不足，但无法掩盖它的优势，当今的许多软件仍在使用C语言进行开发。大家需要经过长时间的开发积累，记住C语言的这些特点，才能更灵活地应用C语言进行开发。

问题5 C语言的应用领域有哪些？

问题阐述

C语言被称为是使用最广泛的高级语言，那么C语言能够应用到哪些领域呢？

专家解答

因为C语言具有高级语言的特点，又具有汇编语言的特点，所以可以作为工作系统设计语言，编写系统应用程序，也可以作为应用程序设计语言，编写不依赖计算机硬件的应



Note



用程序。其应用范围极为广泛，不仅仅是在软件开发上，各类科研项目也都要用到 C 语言。下面列举了 C 语言一些常见的领域。

(1) 应用软件。Linux 操作系统中的应用软件都是使用 C 语言编写的，因此这样的应用软件安全性非常高。

(2) 对性能要求严格的领域。一般对性能有严格要求的地方都是用 C 语言编写的，比如网络程序的底层和网络服务器端底层、地图查询等。

(3) 系统软件和图形处理。C 语言具有很强的绘图能力和可移植性，并且具备很强的数据处理能力，可以用来编写系统软件、制作动画、绘制二维图形和三维图形等。

(4) 数字计算。相对于其他编程语言，C 语言是数字计算能力超强的高级语言。

(5) 嵌入式设备开发。手机、PDA 等时尚消费类电子产品相信大家都不陌生，其内部的应用软件、游戏等很多都是采用 C 语言进行嵌入式开发的。

(6) 游戏软件开发。游戏大家更不陌生，很多人就是由玩游戏而熟悉了计算机。利用 C 语言可以开发很多游戏，比如推箱子、贪吃蛇等。

专家点评

上面仅列出了几个主要的 C 语言应用领域，实际上，C 语言几乎可以应用到程序开发的任何领域。

问题 6 什么是 C99 标准？与 C89 标准相比，C99 标准有哪些新特性？

问题阐述

在编写 C 语言程序的时候，接触比较多的是 C89 标准的编译器，那么什么是 C99 标准呢？它与 C89 标准相比，又有哪些新特性呢？

专家解答

1. 什么是 C99

本书涉及的大部分 C 语言技术知识都是 ANSI C 标准实现的，ANSI C 标准是在 1989 年提出的，故通常称之为 C89 标准。在 ANSI 标准化后，C 语言的标准在相当长的一段时间内都保持不变，尽管 C++ 在不断地改进（实际上，Normative Amendment1 在 1995 年已经开发了一个新的 C 语言版本，但是这个版本很少为人所知）。这一局面直到 1999 年才有所改变，这就是 ISO9899:1999 的问世。该版本就是通常提及的 C99，它被 ANSI 于 2000 年 3 月采用。其实在初学阶段 C89 和 C99 的区别是不易察觉的，所以有许多人还没有注意到 C99。下面就来了解一下 C99 新修改了哪些方面的内容。



Note



C99 标准是在 ANSI C (C89) 标准的基础上发展起来的, 较 C89 增加了许多方面的内容, 例如基本数据类型、关键字和一些系统函数等。

2. C99 有哪些新特性

C99 和 C89 的大多数特性基本相同, 差别很小。在 C99 中, 其新增特性简单地归结为以下几点。

(1) 对编译器限制增加了, 比如源程序每行要求至少支持 4095 字节, 变量名、函数名要求支持 63 字节。

(2) 支持//行注释。

例如:

```
#include<stdio.h>
main()
{
    int i,j;                /*定义变量 i 和 j*/
    i=90;                   /*给变量 i 赋值 90*/
    j=i*i;                  /*j 等于 i*i*/
    printf("%d",j);         /*将 j 的值输出*/
}
```

上面的注释使用的是/* */, 在 C99 标准下也可以将注释写成如下的形式(大部分编译器支持)。

```
#include<stdio.h>
main()
{
    int i,j;                //定义变量 i 和 j
    i=90;                   //给变量 i 赋值 90
    j=i*i;                  //j 等于 i*i
    printf("%d",j);         //将 j 的值输出
}
```

(3) 增加了新关键字 restrict、inline、_Complex、_Imaginary、_Bool。

- ☑ restrict: 该关键字只用来修饰指针。表明 restrict 类型的指针是访问一个数据对象的唯一且初始的方式, 其他指针必须要基于第一个指针才能对对象进行存取。restrict 指针主要用作函数变元, 或者指向由 malloc()函数所分配的内存变量。restrict 数据类型不改变程序的语义。
- ☑ inline: 该关键字用在函数之前, 表示该函数为内联函数, 程序在编译的时候直接扩展为函数的代码, 而不是调用函数。内联函数除了保持结构化和函数式的定义方式外, 还能使程序员写出高效率的代码。函数的每次调用与返回都会消耗相当多的系统资源, 尤其是当函数调用发生在重复次数很多的循环语句中时。一般情况下, 当发生一次函数调用时, 变元需要进栈, 各种寄存器内存需要保存。当函数返回时, 寄存器的内容需要恢复。如果该函数在代码内进行联机扩展, 当代码



Note



Note

执行时，这些保存和恢复操作会再次发生，而且函数调用的执行速度也会大大加快。函数的联机扩展会产生较长的代码，所以只应该内联对应用程序性能有显著影响的函数以及长度较短的函数。

- ☑ **_Bool**: 布尔类型，该类型有两个常量 `true` 和 `false`。C99 中增加了用来定义 `bool`、`true` 以及 `false` 宏的头文件 `<stdbool.h>`，以便程序员能够编写同时兼容于 C 与 C++ 的应用程序。在编写新的应用程序时，应该使用 `<stdbool.h>` 头文件中的 `bool` 宏。
- ☑ **_Complex 和 _Imaginary**: C99 标准中定义的复数类型如下。
 - `float_Complex`。
 - `float_Imaginary`。
 - `double_Complex`。
 - `double_Imaginary`。
 - `long double_Complex`。
 - `long double_Imaginary`。

`<complex.h>` 头文件中定义了 `complex` 和 `imaginary` 宏，并将它们扩展为 `_Complex` 和 `_Imaginary`，因此在编写新的应用程序时，应该使用 `<stdbool.h>` 头文件中的 `complex` 和 `imaginary` 宏。

(4) 支持 `long long`、`long double_Complex`、`float_Complex` 这样的类型。

(5) 支持不定长的数组。这样定义数组时，数组长度就可以使用变量了。在 C99 中，程序员声明数组时，数组的维数可以由任一有效的整型表达式确定，包括只在运行时才能确定其值的表达式。这类数组就叫做可变长数组，但是只有局部数组才可以是可变长的。可变长数组的维数在数组生存期内是不变的，也就是说，可变长数组不是动态的，可以变化的只是数组的大小，可以使用 “*” 来定义不确定长的可变长数组。声明类型的时候就可以用如下形式。

```
int a[*];
```

这样的写法不能用在全局或者共用体中。

(6) 对于数组声明中的类型修饰符，在 C99 中，如果需要使用数组作为函数变元，可以在数组声明的方括号内使用 `static` 关键字，这相当于告诉编译程序，变元所指向的数组将至少包含指定的元素个数。也可以在数组声明的方括号内使用 `restrict`、`volatile`、`const` 关键字，但只用于函数变元。如果使用 `restrict`，指针是初始访问该对象的唯一途径。如果使用 `const`，指针始终指向同一个数组。使用 `volatile` 没有任何意义。

(7) 变量声明不必放在语句块的开头，`for` 语句可以写成如下形式。

```
for(int i=0;i<100;++i)
```

不需要将变量 `i` 的声明格外提出来写在外面。这样做 `i` 在 `for` 语句里一直是有效的，`i` 在 `for` 语句外面是否有效根据编译器的不同而不同。



(8) 当一个类似结构的东西需要临时构造的时候, 可以用

```
(type_name)
{
    ...
}
```

(9) 在字符串中, \u 支持 Unicode 的字符。

(10) 支持十六进制的浮点数的描述, 所以 printf、scanf 的格式化串支持 ll/LL 对应新的 long long 类型。在 printf() 和 scanf() 函数中, ll 适用于 d、i、o、u 和 x 格式声明符。另外, C99 还引进了 hh 修饰符。当使用 d、i、o、u 和 x 格式声明符时, hh 用于指定 char 型变元。ll 和 hh 修饰符均可以用于 n 声明符。

(11) 使用 A 格式修饰符时, x 和 p 必须是大写。A 和 a 格式修饰符也可以用在 scanf() 函数中, 用于读取浮点数。调用 printf() 函数时, 允许在 %f 声明符前加上 l 修饰符, 即 %lf, 但不起作用。

(12) 预处理的修改: 预处理中宏可以设定参数, 宏定义中用 (...) 表示, 预处理标识符 _VA_ARGS_ 表示在什么情况下替换参数。例如:

```
#define ClassAVG(...) avg(_VA_ARGS_)
```

(13) 浮点数的内部数据描述支持新标准, 这可以用 #pragma 编译器指定。

(14) 除了已经有的 _line_ 和 _file_ 以外, 还支持 _func_, 用于得到当前的函数名。

(15) 对于有些敏感数的表达式, 也允许编译器进行相应化简。

(16) 取消了函数必须有返回值这一条。原来 C89 中规定如果函数没有指定类型, 则默认为 int 类型。

(17) 允许 struct 定义的最后一个数组写为 [], 不指定其长度描述, 这就叫做柔性数组成员, 但结构中的柔性数组成员前面必须至少有一个其他成员。柔性数组成员允许结构中包含一个大小可变的数组。sizeof 返回的这种结构大小不包括柔性数组的内存。包含柔性数组成员的结构用 malloc() 函数进行内存的动态分配, 并且分配的内存应该大于结构的大小, 以适应柔性数组的预期大小。

(18) 对 const const 的化简。例如:

```
const const int x;
```

将被当作

```
const int x;
```

处理。

(19) C99 引入了在程序中定义编译指令的另外一种方法: _Pragma 运算符。格式如下:

```
_Pragma("directive")
```

其中 directive 是编译指令。_Pragma 运算符允许编译指令参与宏替换。



Note



(20) C99 新增了头文件, 如表 1.1 所示。

表 1.1 新增头文件

头 文 件	用 途
<complex.h>	支持复数算法
<fenv.h>	给出对浮点状态标记和浮点环境的其他方面的访问
<inttypes.h>	定义标准的、可移植的整型类型集合, 也支持处理最大宽度整数的函数
<iso646.h>	于1995年第一次修订时引进, 用于定义对应各种运算符的宏
<stdbool.h>	支持布尔数据类型, 定义宏bool, 以便兼容于C++
<stdint.h>	定义标准的、可移植的整型类型集合, 该文件包含在<inttypes.h>中
<tgmath.h>	定义一般类型的浮点宏
<wchar.h>	于1995年第一次修订时引进, 用于支持多字节和宽字节函数
<wctype.h>	于1995年第一次修订时引进, 用于支持多字节和宽字节分类函数

专家点评

当前各编译器大多支持 C89 标准。在使用编译器时, 应该注意编译器遵守的是 C89 标准还是 C99 标准, 只有按照编译器的标准编写程序, 才不会使编译出现问题。

问题 7 C 语言是 C++ 的子集吗?

问题阐述

C 语言是 C++ 的子集吗? C++ 是在 C 语言的基础上扩展而来并包含所有 C 语言的内容吗?

专家解答

从实用角度讲, C++ 属于 C 语言的一个超集, 基本上兼容 ANSI C。但是从编译角度上讲, C 语言的有些特性在 C++ 中并不支持。相反, ANSI C 继承了 C++ 的几个特性, 包括原型和常量。因此, 这两种语言并不是另一个的超集或子集; 而且它们在一些通用构造的定义上也不同。尽管有这些不同, 许多 C 程序在 C++ 环境中仍能编译, 而且许多最新的编译器同时提供 C 和 C++ 的编译模式。但是, 不要把 C 代码完全当做 C++ 代码来编译, 否则在遇到不兼容问题时会给程序带来错误。

C++ 对 C 语言的改进如下。

- ☑ C++ 对数组定义进行了限制。在 C 语言中, 初始化数组时不作数组溢出判断, 这样就容易使数组没有足够大的空间存放数据而产生错误。C++ 对此作了一些改进, 像 `char str[3]="Jim"` 这样的表达式就被认为是一个错误, 但是它在 C 语言中是合法的。
- ☑ 在 C++ 中, 声明语句可以穿插于语句之间。大家知道, 在 C 语言中, 一个语句块



中的所有声明必须都放在所有语句的前面，而 C++ 去掉了这个限制，使声明语句可以穿插于语句之间。

- ☑ C++ 对 C 语言的改进最主要表现在对面向对象的扩充上。C 语言是一种面向过程的结构化的语言，而 C++ 是面向对象的语言，它在 C 语言的基础上增加了面向对象的机制，使得 C++ 比 C 语言更加完善和实用。

C++ 中存在而 C 语言中不存在的限制。

- ☑ 在 C++ 中，用户代码不能够调用主函数 `main()` 函数，但是在 C 语言中这是可以的（但是极少出现这种情况）。
- ☑ C++ 中对函数原型的声明是严格的，要求必须完整，而在 C 语言中却没有这么严格。
- ☑ 在 C++ 中，由 `typedef` 定义的类型的名词不能与已有的结构名称冲突，但在 C 语言中却是允许的。
- ☑ C++ 规定了更严格的类型处理，例如，当 `void*` 指针赋值给另一个类型的指针时，C++ 要求进行强制类型转换，而在 C 语言中则无需这么做。

C++ 与 C 语言中含义不同的特性：

- ☑ C++ 对 C 语言的关键字进行了扩充，增加了至少十几个。这些关键字在 C 语言中可以作为标识符使用，但是如果这样的 C 语言代码使用 C++ 编译器进行编译，就会产生错误信息。
- ☑ 在 C++ 中，内层作用域的结构名称将会隐藏外层空间中相同的对象名，在 C 语言中则不会这样。
- ☑ 在 C++ 中，注释可以使用 “//” 注释符，而在 C 语言中则不能使用。

专家点评

从总体上说，C 语言是 C++ 的前身，上面只介绍了它们的一部分不同之处。从上面的介绍可以看出，C++ 并没有对 C 语言存在的一些最基本的问题进行改进，它仍然保留了 C 语言的许多缺陷，而且在此基础上又堆积了大量复杂的东西。不过尽管存在着不少缺陷，C 语言和 C++ 都被广泛地使用着，这也说明了其存在的价值。

C 和 C++ 的不同之处还有很多，大家要通过实际编程来分析两者的差异，并保持警惕，避免相互使用时出现错误。

问题 8 C 语言程序的开发过程是怎样的？

问题阐述

C 语言程序的整个开发过程是怎样的？都包括哪些步骤？



Note



专家解答

C 语言程序的开发主要包括编辑、编译、连接和运行 4 个步骤。

(1) 编辑源文件：利用 C 语言编译系统自带的编辑器（例如，TC 2.0 的代码编辑器）或者其他程序编辑器将编写的 C 语言程序代码输入计算机，然后将编辑好的代码保存在磁盘中，这样就生成了 C 语言的源文件。

(2) 编译源文件：利用编译器将源程序翻译成目标代码。其具体过程是先将源程序转换成汇编语言程序，然后再将汇编语言程序翻译成机器指令程序，即目标程序。

(3) 连接：将目标程序的各个部分进行连接配置，生成一个可供运行的可执行程序。

(4) 运行：将编译好的可执行文件运行。通过运行程序可以查看程序执行输出结果。

C 语言程序代码的编译和运行过程如图 1.1 所示。

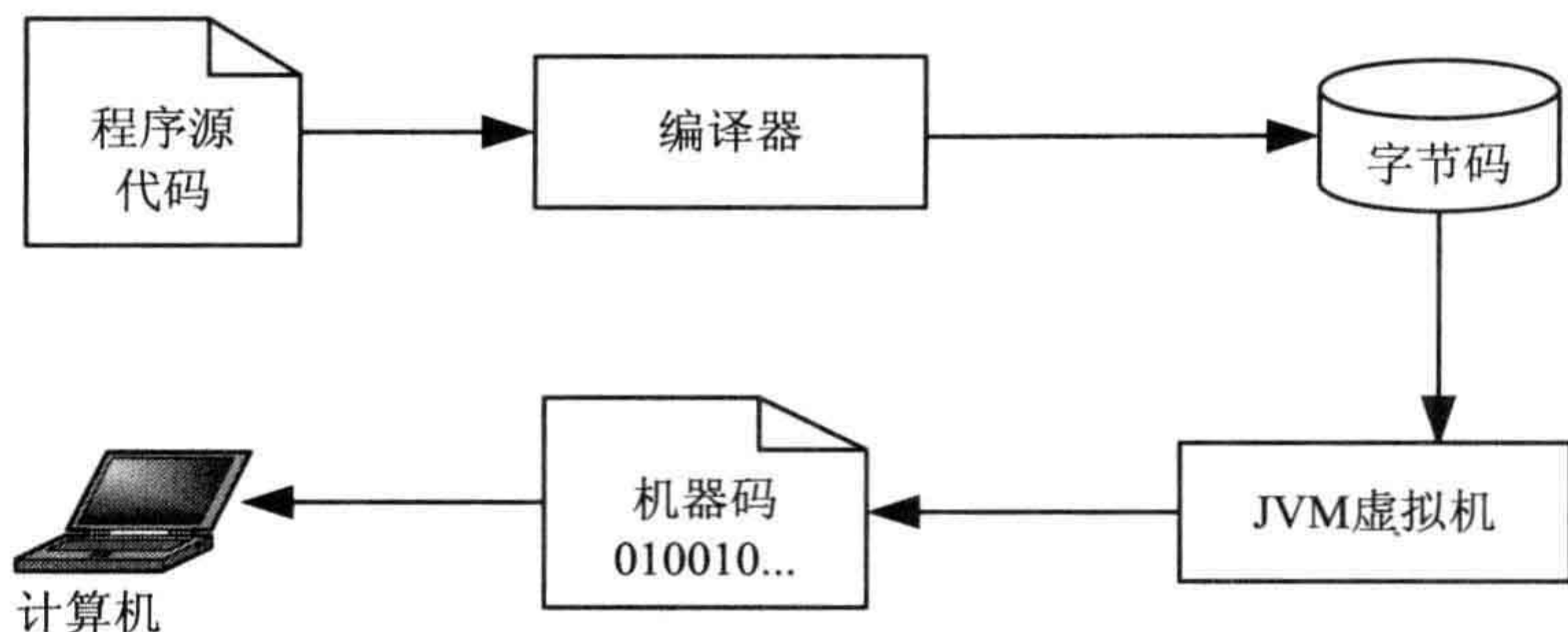


图 1.1 C 语言程序的编译和运行过程

专家点评

在学习的时候，大家只需要掌握基本的程序设计步骤即可，具体如何实现编译和代码转换等操作则不需要深究（如果有兴趣，可以进行深入的学习）。

问题 9 什么是编译程序和解释程序？

问题阐述

什么是编译程序和解释程序？

专家解答

编译程序和解释程序是实现程序运行的途径。原则上说，任何语言都可以编译，也可以解释，但往往很多语言只通过编译和解释就能运行，C 语言就是其中一种，它通过编译实现运行。

C 语言的编译程序将源程序转换为目标代码，然后由计算机直接运行。编译过程本身



占用了一些额外的时间，但是当运行程序时，这一点很容易得到补偿。经过编译的程序比在解释环境下运行的程序快很多；唯一的例外是程序特别短，不足 50 行，且没有循环语句。

专家点评

不同的编译器所执行的标准有所不同，所以在编写代码的时候要考虑到编译器执行的标准，从而保证编写出正确的代码。



Note

问题 10 ANSI C 的编译限制有哪些？

问题阐述

ANSI C 标准对 C 程序作了很多限制，那么大体上有哪些呢？

专家解答

编译标准都会对程序作出一些限制，而且大多数编译器的规定大体类似，如一个标识符最大可以有多少个字符、一个数组的维数最大可以达到多少等。对于 ANSI C 编译器，其至少要能够支持以下条件。

- ☑ 在函数定义中形参数量上限至少可以达到 31 个。
- ☑ 在函数调用时实参数量上限至少可以达到 31 个。
- ☑ 在一行源代码中至少可以有 509 个字符。
- ☑ 在一个表达式中至少可以支持 32 层嵌套的括号。
- ☑ 长整型 (long int) 数据的最大值不得小于 2147483647 (即不得低于 32 位)。

专家点评

这里只介绍了一部分 ANSI C 标准对编译器的限制。一个遵循标准的编译器，必须能够编译并执行一个满足上面这些限制的程序。但是在程序代码违反上面这些限制时，编译器不一定产生错误信息。

第 2 章

一个简单的 C 程序

- ▶▶ C 语言的入口函数是什么？
- ▶▶ C 语言程序由哪些部分组成？
- ▶▶ 如何在 Turbo C 2.0 中输入一个程序？
- ▶▶ 如何在 Visual C++ 6.0 中运行一个 C 程序？
- ▶▶ 如何在 Visual C++ 2008 中运行一个 C 程序？
- ▶▶ 如何提高程序的可读性？
- ▶▶ 什么是关键字？C 语言的关键字有哪些？
- ▶▶ 什么是标识符？使用标识符的注意事项是什么？
- ▶▶ void 关键字都有哪些用途？
- ▶▶ 什么是匈牙利命名约定？它是否是好的约定？



问题 11 C 语言的入口函数是什么?

问题阐述

C 语言程序是如何调用执行的? C 语言的入口函数是什么?

专家解答

在 C 语言中, `main()` 函数称为主函数, 作为程序的入口函数。程序的执行从 `main()` 函数开始, 调用其他函数后流程返回到 `main()` 函数, 最后在 `main()` 函数中结束整个程序的运行。`main()` 函数是系统定义的, 也是由系统调用的。

每一个 C 语言程序都必须有一个且只能有一个 `main()` 函数。函数体由大括号 `{}` 括起来。`main()` 函数可以定义为下面的形式:

无参数形式:

一般使用的 `main()` 函数都是无参数的, 由系统直接调用。其定义格式为:

```
int main()
int main(void)
```

有参数形式:

在运行程序时, 有时需要将必要的参数传递给主函数, 主函数 `main()` 的形式参数如下。

```
main (int argc, char* argv[])
```

两个特殊的内部形参 `argc` 和 `argv` 是用来接收命令行实参的, 这是只有主函数 `main()` 才能具有的参数。

(1) `argc` 参数。`argc` 参数保存命令行的参数个数, 是个整型变量。这个参数的值至少是 1, 因为至少程序名就是第一个实参。

(2) `argv` 参数。`argv` 参数是一个指向字符指针数组的指针, 在这个数组里的每一个元素都指向命令行实参。所有命令行实参都是字符串, 任何数字都必须由程序转变为适当的格式。

专家点评

`main()` 函数是 C 语言程序的入口函数, 也是一个 C 语言程序中必不可少的函数。如果程序中没有 `main()` 函数, 则程序将无法运行。控制台程序的入口是 `main`, Win32 程序的入口是 `WinMain`。在开发时要了解这个特点。



问题 12 C 语言程序由哪些部分组成?



问题阐述

一个 C 语言程序都由哪些部分组成? 它的基本单位是什么?

专家解答

一个 C 语言程序可以由一个主函数和若干个函数构成。一个大的应用程序一般应该分为多个程序模块, 每一个模块用来实现一个功能。实现这些模块功能的可以叫做子程序。在 C 语言中, 模块的功能是由函数完成的。通常用 tc 写的程序也就一个文件, 但是用 C 语言写大程序的时候就不能把所有代码都写在一个文件中, 要写到很多个文件中。这样可以分别编写、分别编译, 以此提高调试效率, 同时增加 C 程序模块的可移植性。一个源文件可以被多个 C 程序公用。

一个文件可以称为一个源程序文件, 一个源程序文件由一个或者多个函数组成。在 C 语言中, 函数是组成程序的最小单位。一个源程序文件是一个编译单位, 即编译器是以源程序为单位进行编译的, 而不是以函数为单位进行编译的。

专家点评

在编写比较大的程序时, 要善于将代码分类, 将常用功能模块编写成函数, 放在函数库中供公共调用, 要善于利用文件和函数, 以减少代码的重复编写。

问题 13 如何在 Turbo C 2.0 中输入一个程序?

问题阐述

C 程序最经常使用的编辑器就是 Turbo C。如何在 Turbo C 2.0 中输入一个程序? 具体的过程是怎样的?

专家解答

为了能使用 Turbo C, 必须先将 Turbo C 编译程序装入磁盘的某一目录下, 例如, 放在 C 盘根目录下一级 TC 子目录下。

(1) 打开 tc.exe, 屏幕上将出现 Turbo C 集成环境, 如图 2.1 所示。

(2) 通过图 2.1 可以看到, 在集成环境的上部有一行“主菜单”, 这 8 个菜单项分别代表: 文件操作、编辑、运行、编译、项目文件、选项、调试、中断/观察等功能。按键盘上的左右键可以进行菜单间的切换, 被选中的项以“反相”形式显示。此时按 Enter 键, 就会出现一个下拉菜单, 通过上下键可以选择所需要的项。例如, 要打开 D:\pro\12.C 文件



的过程如图 2.2 所示。



图 2.1 Turbo C 集成环境

(3) 编辑源文件。在编辑状态下，可以根据需要输入或修改源程序。

(4) 编译源程序。选择 Compile 菜单，并在其下拉菜单中选择 Compile to OBJ 项，则进行编译，得到一个后缀为.obj 的目标程序，如图 2.3 所示。

(5) 进行编译后，再选择 Link EXE file 项进行连接操作，可得到一个后缀为.exe 的可执行文件，如图 2.4 所示。

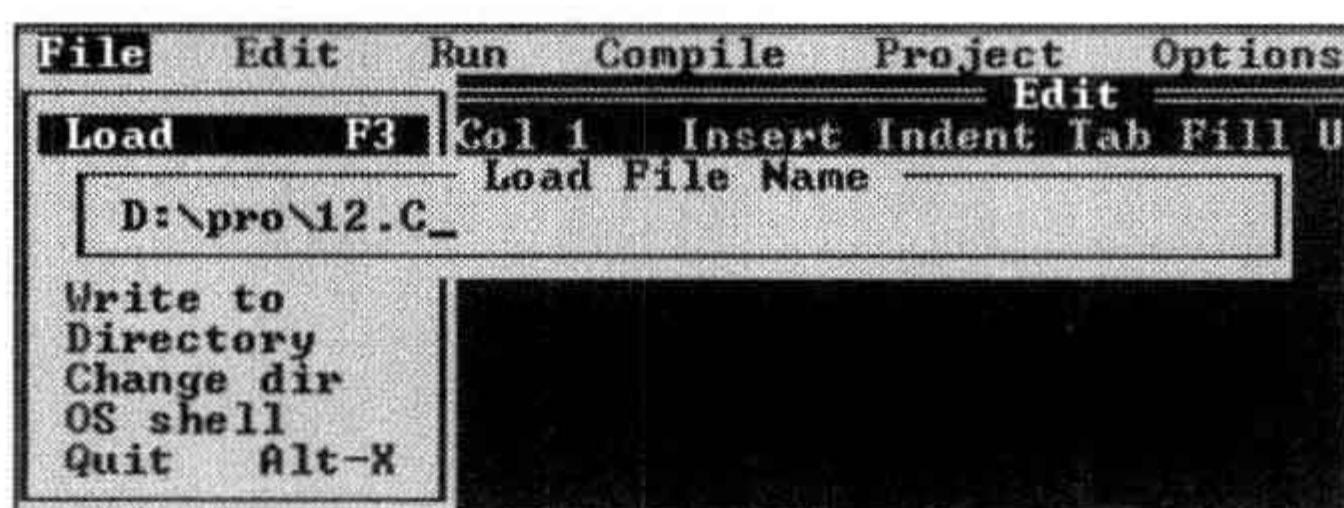


图 2.2 打开指定文件

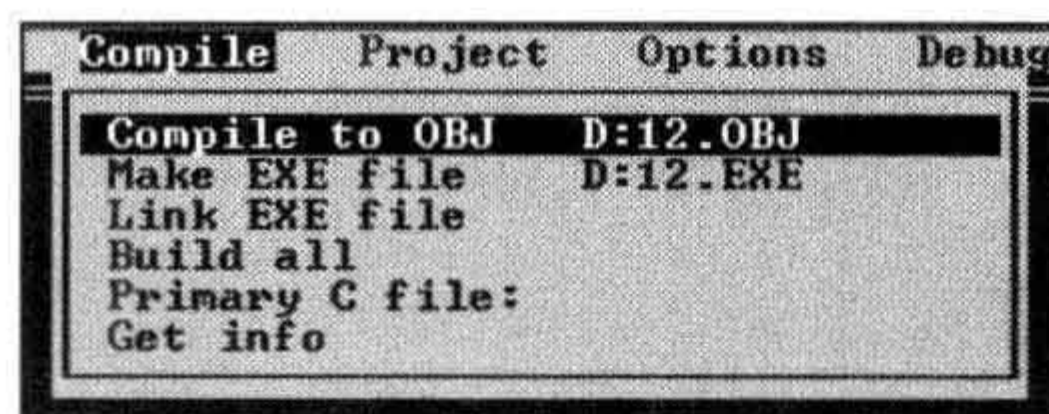


图 2.3 编译源程序

(6) 这里也可以将上面两个步骤合并成一个步骤进行，选择 Make EXE file 项或按 <F9>键，就可以一次完成编译和连接操作，如图 2.5 所示。

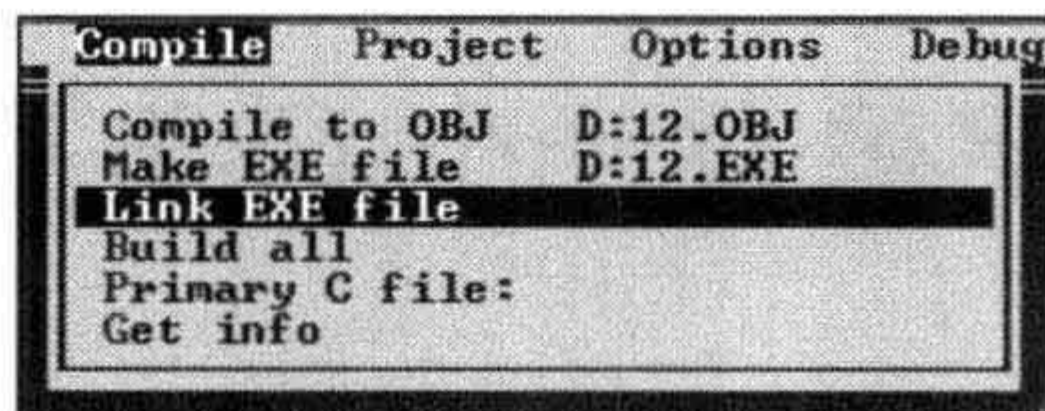


图 2.4 连接

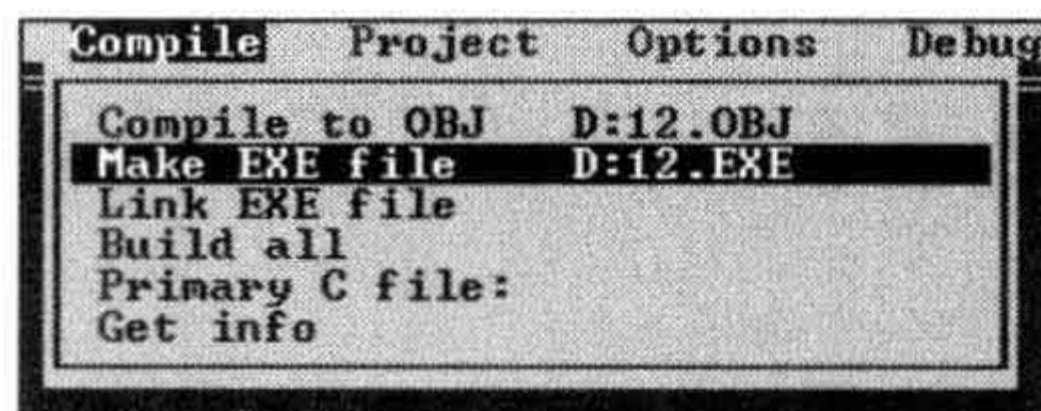


图 2.5 编译连接

(7) 执行程序。选择 Run 下拉菜单中的 Run 项，或直接按 <Ctrl+F9>组合键，系统便会执行已编译和连接好的目标文件。

如果运行后出现错误，需要修改源程序，这时按 <Alt+E>组合键就重新回到编辑状态。

(8) 退出 Turbo C 环境。可以选择 File 下拉菜单中的 Quit 项，也可按 <Alt+X>组合键退出。退出时应对文件进行保存，对未保存的文件系统会给出提示信息。

注意：

在进行文件保存时，要注意文件扩展名的修改。



当 Turbo C 编译程序没有放在 C 盘根目录下一级 TC 子目录下, 而是放在 D 盘根目录下一级 TC 子目录下时, 在对源文件进行编译和连接前, 应改一下路径, 具体操作步骤如下:

选择 Options 下拉菜单中的 Directories 项, 如图 2.6 所示。

将下拉菜单中的路径改为当前目录, 即 D 盘根目录下一级 TC 子目录, 如图 2.7 所示。

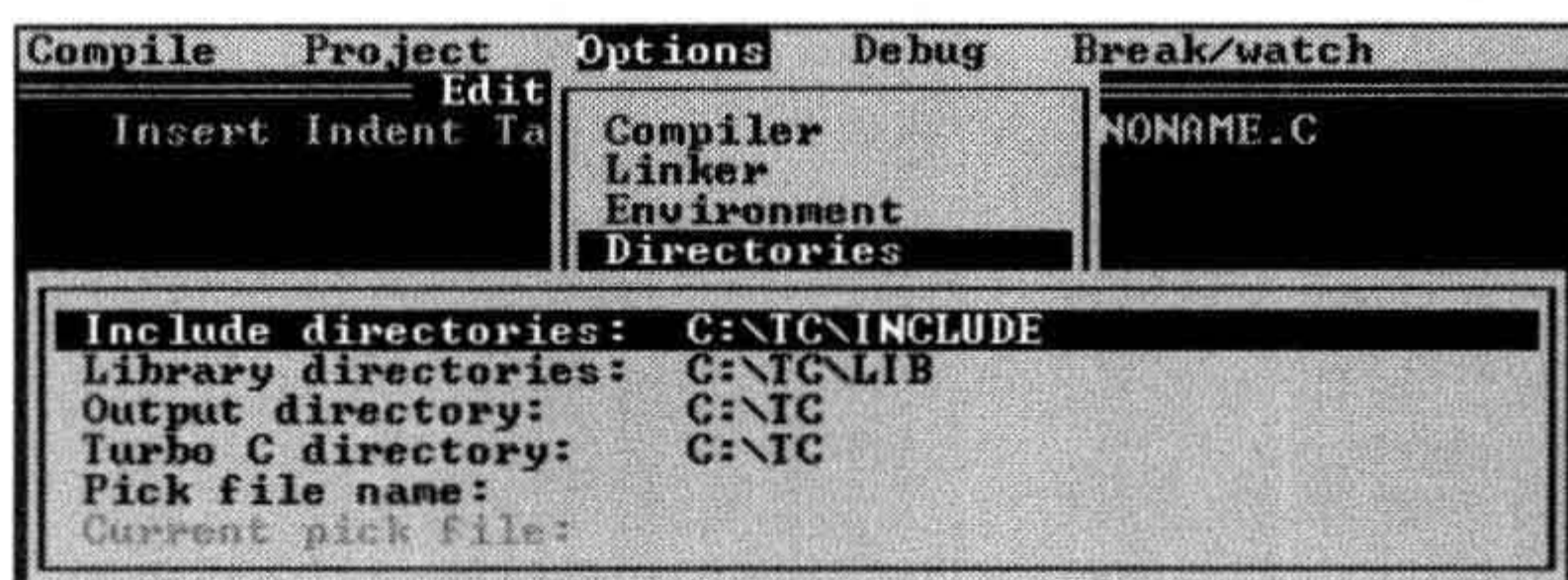


图 2.6 Options/Directories

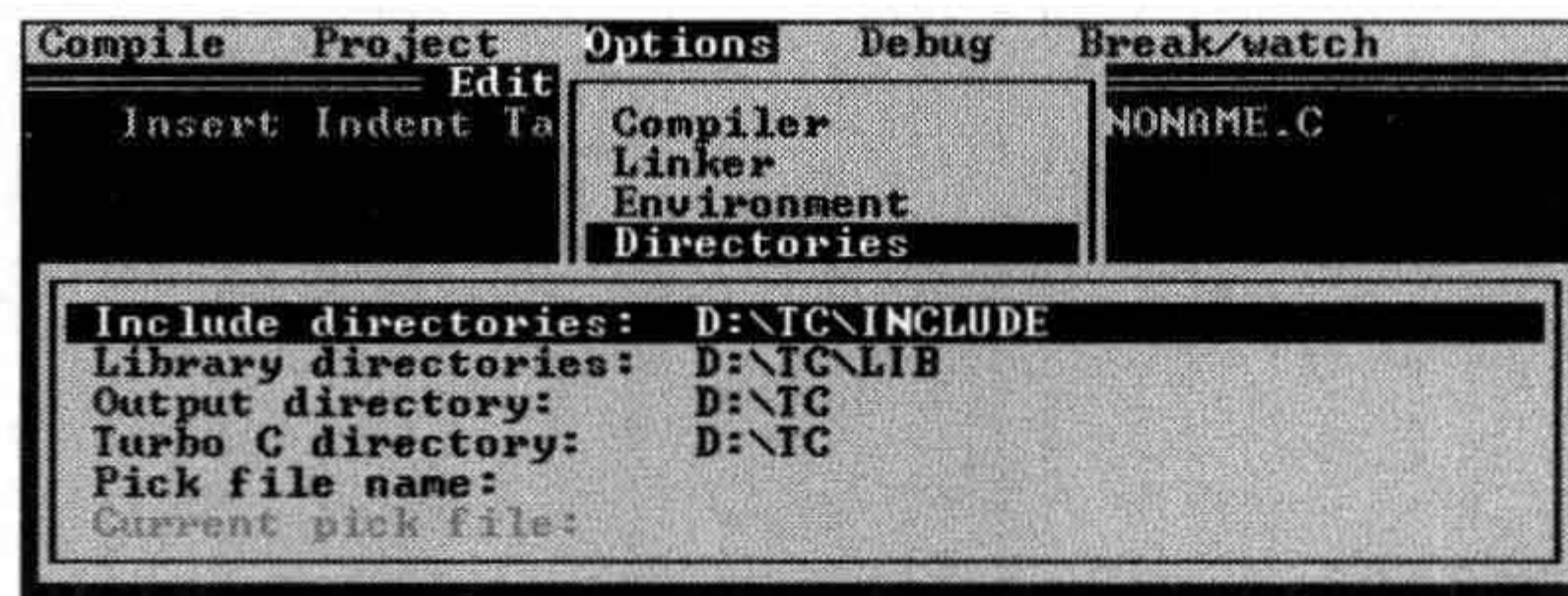


图 2.7 修改后的路径

在修改完路径后, 还要对修改的路径进行保存, 即选择 Options 下拉菜单中的 Save options 项, 如图 2.8 所示。如果源文件已经存在, 则提示是否覆盖, 如图 2.9 所示。

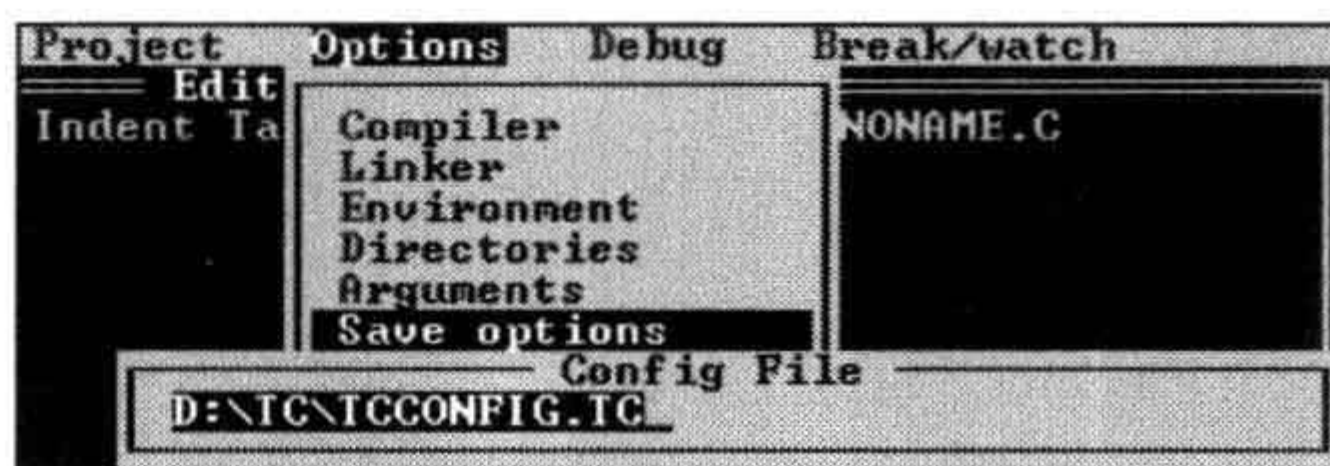


图 2.8 配置文件

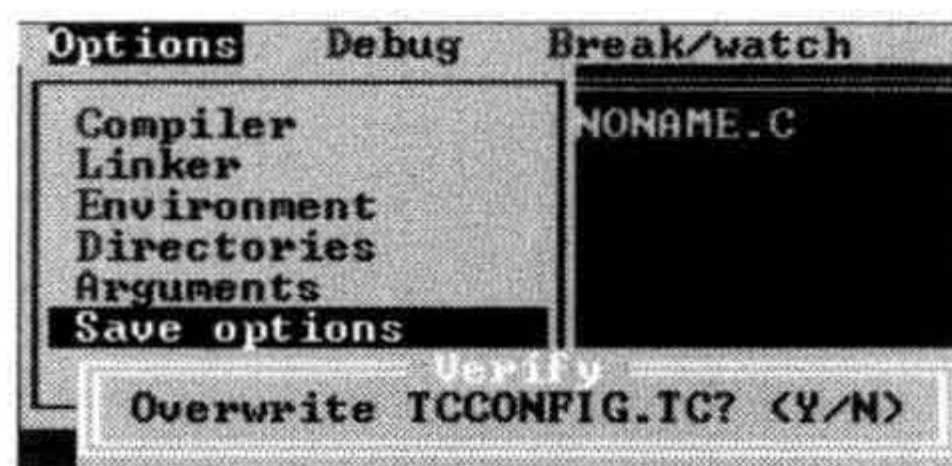


图 2.9 保存路径

专家点评

Turbo C 是广泛使用的 C 语言编译程序。它有方便、直观的界面和丰富的库函数, 但是 Turbo C 2.0 不支持鼠标操作, 这给操作带来一定的麻烦, 读者可以选择使用高级的版本, 以方便操作。

问题 14 如何在 Visual C++ 6.0 中运行一个 C 程序?

问题阐述

Visual C++ 是 Microsoft Visual Studio 6.0 家族成员之一, 具有强大的可视化开发环境, 为程序员开发软件提供了方便的条件。使用 Visual C++ 6.0 开发环境同样可以开发 C 语言程序。下面介绍如何在 Visual C++ 6.0 中开发一个 C 程序?

专家解答

在 Visual C++ 6.0 中开发一个 C 程序的步骤如下。

(1) 在安装了 Visual C++ 6.0 之后, 通过开始菜单启动 Visual C++ 6.0, 如图 2.10 所示。



图 2.10 启动 Visual C++ 6.0

进入 Visual C++ 6.0 界面内, 如图 2.11 所示。

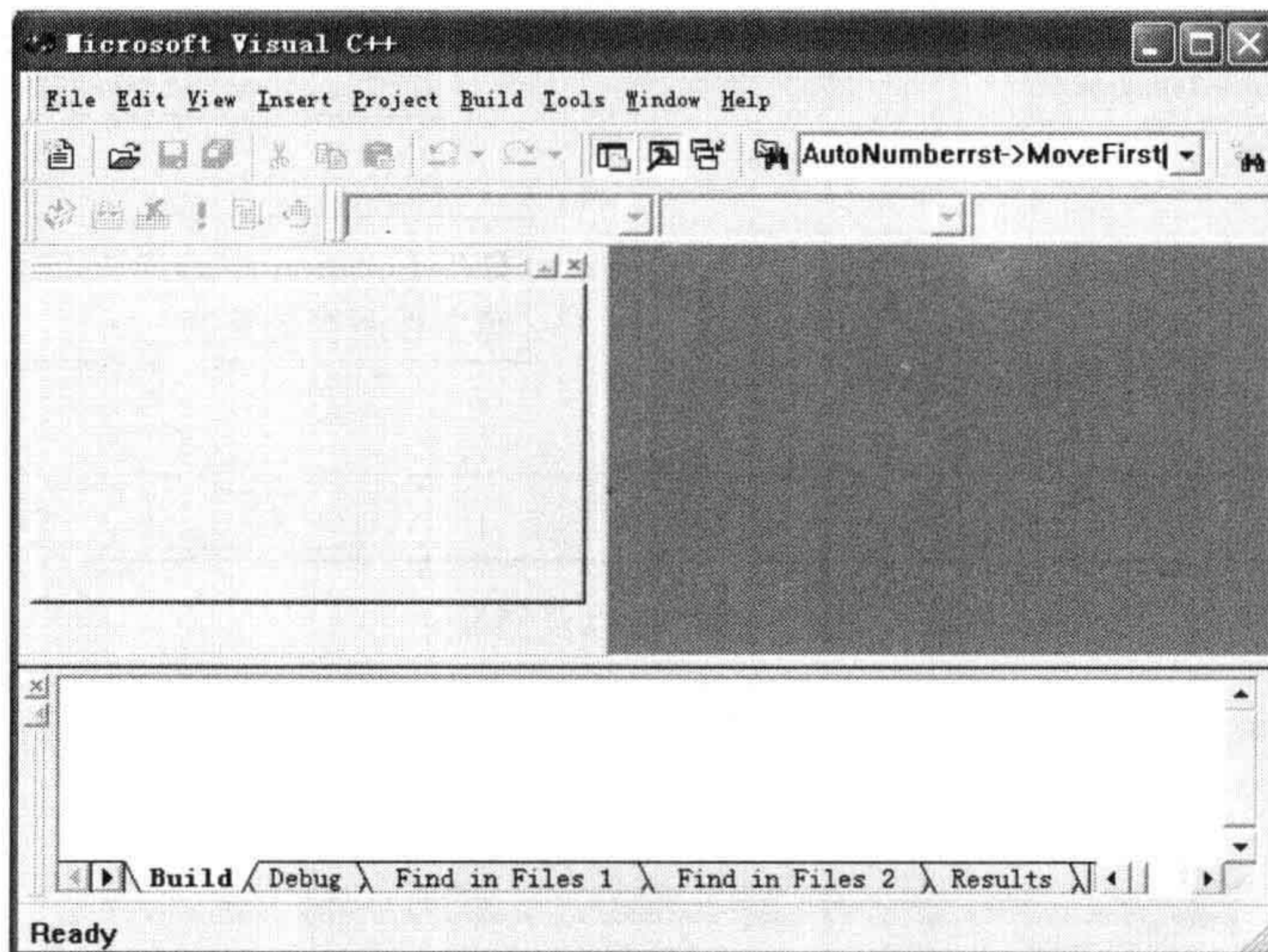


图 2.11 Visual C++ 界面

(2) 在 File 下拉菜单中选择 New 选项, 或者直接使用快捷键<Ctrl+N>, 如图 2.12 所示。

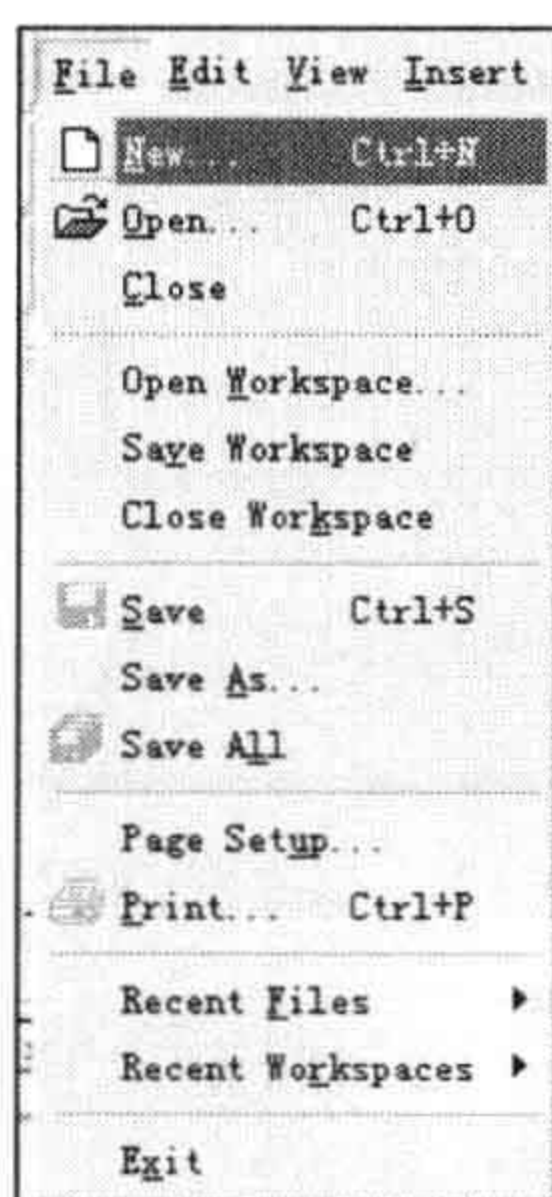


图 2.12 新建项目

(3) 打开如图 2.13 所示界面, 选择 Files 选项卡, 在列表中选择 C++ Source File 选项, 在右侧 File 文本框中填写要创建的 C 文件名称 (如果要创建.c 文件, 则要写上扩展名), 设置创建的文件存放路径。





Note

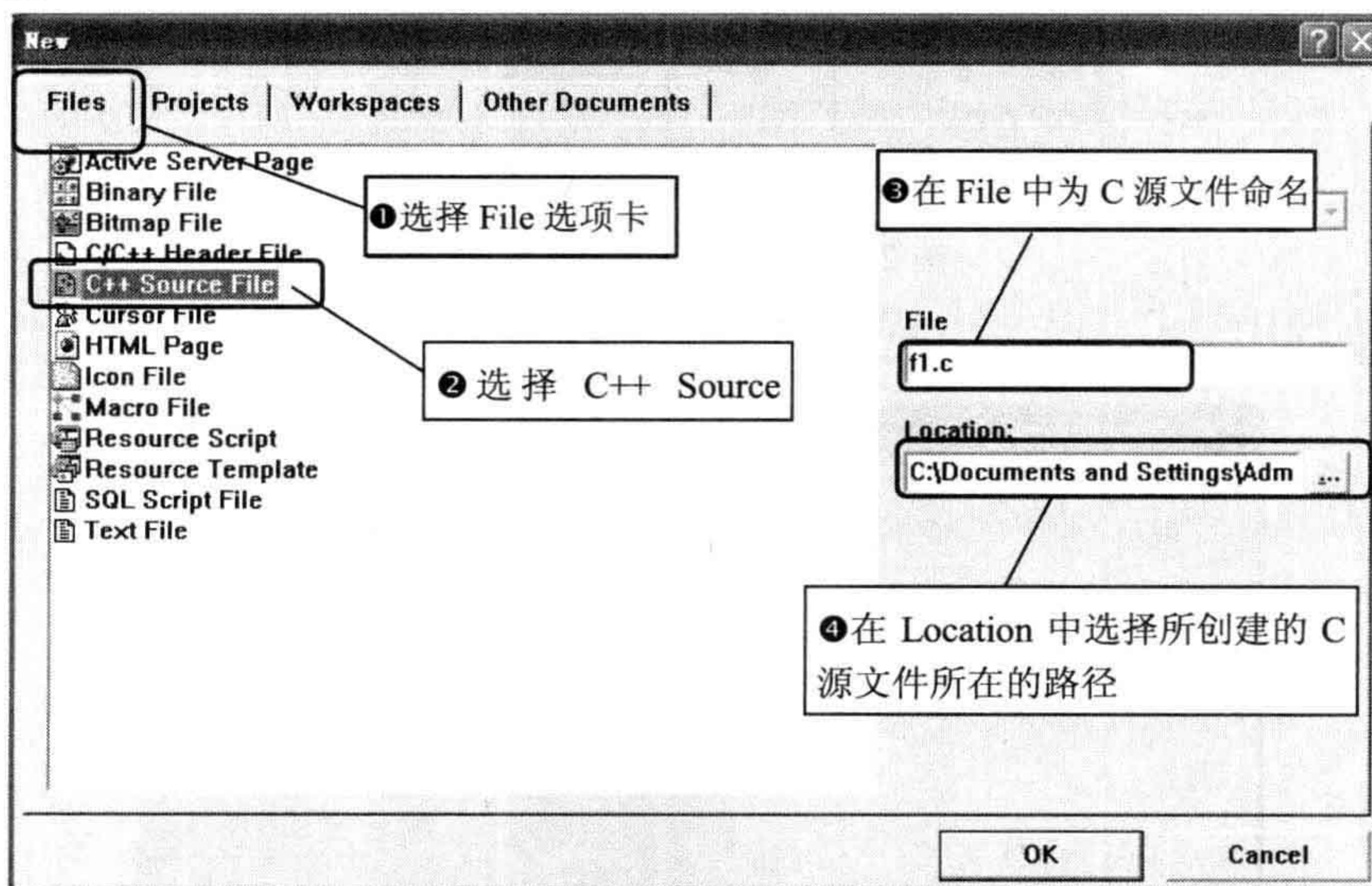


图 2.13 名称及路径

(4) 单击 OK 按钮，打开 C 文件代码编辑区，在代码编辑区编写代码，如图 2.14 所示。

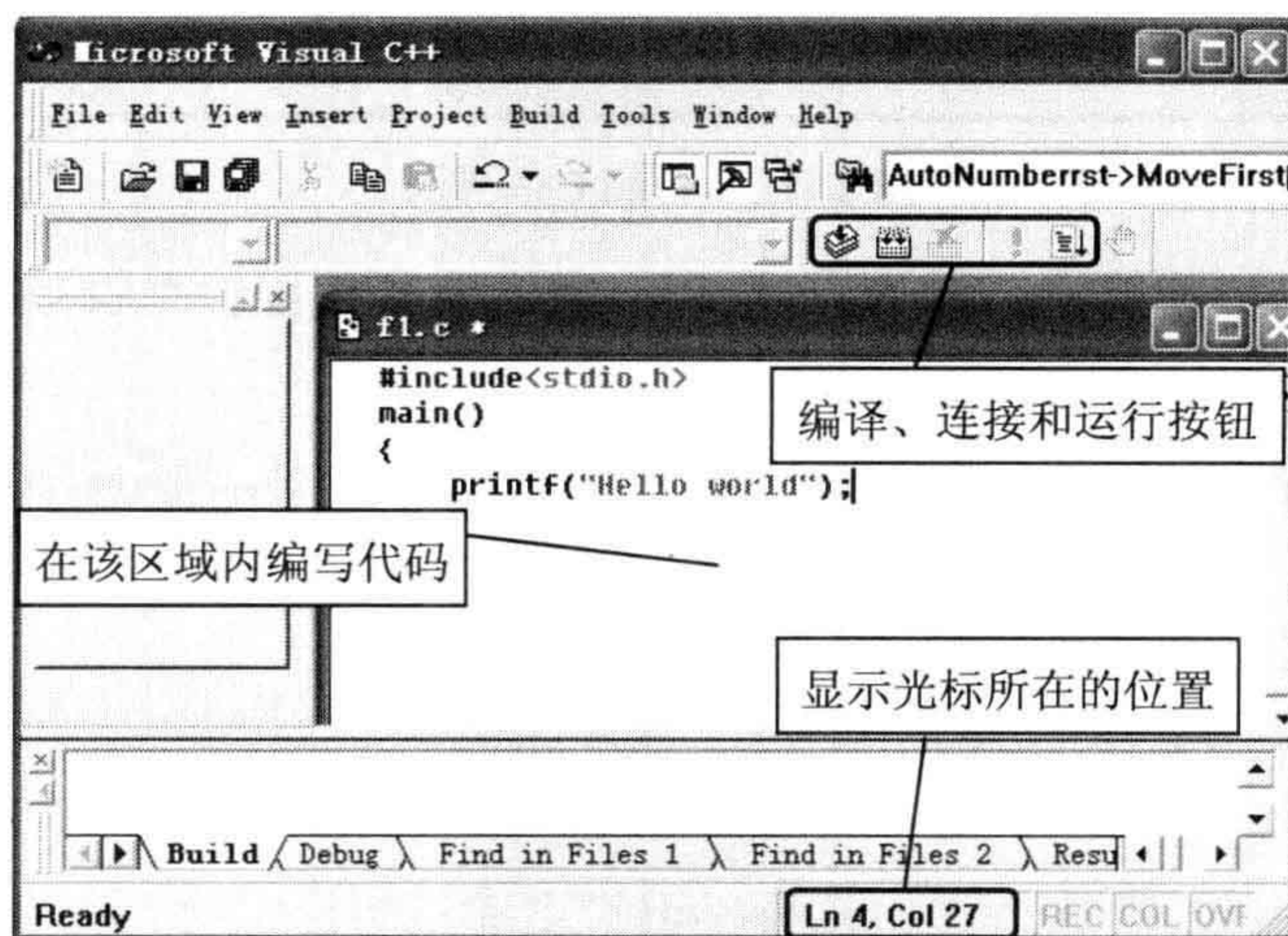


图 2.14 编辑代码界面

(5) 编写好程序代码后，选择 build 下拉菜单中的 Compile f1.c 选项，编译程序，如图 2.15 所示。

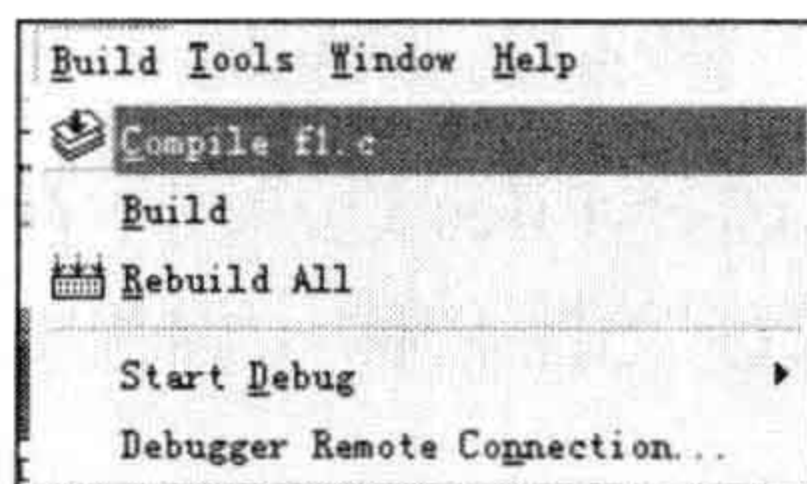


图 2.15 编译



此时会弹出一个对话框，提示是否创建工作区文件，如图 2.16 所示。

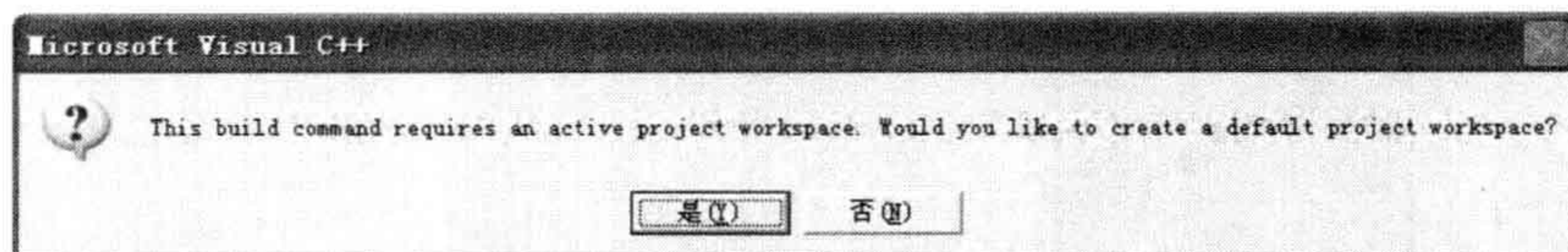


图 2.16 创建工作区

(6) 单击“是”按钮后会再弹出一个对话框，如图 2.17 所示。

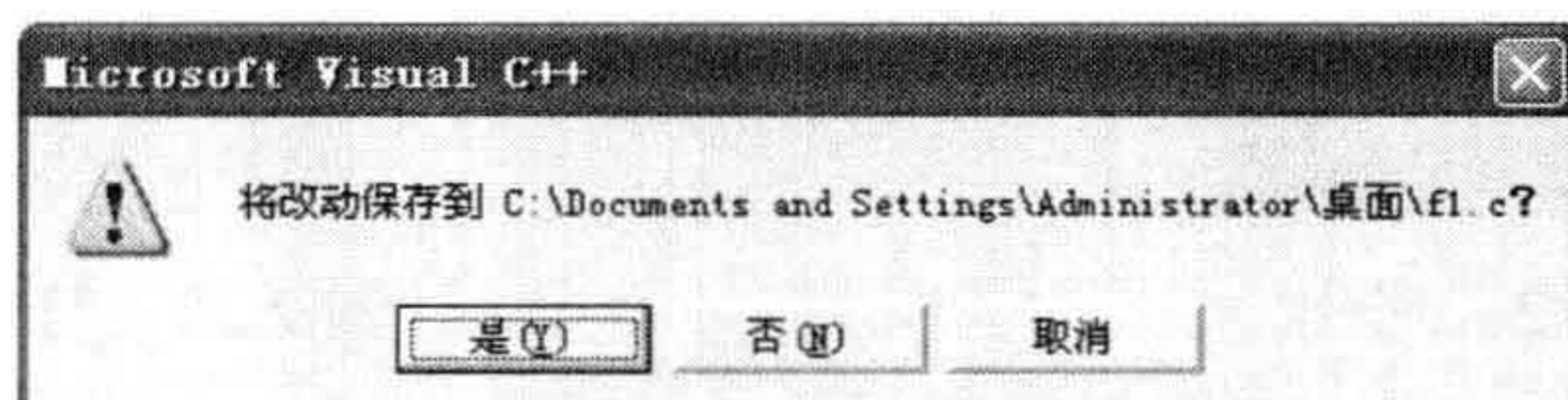


图 2.17 保存文件到指定路径

(7) 单击“是”按钮后选择 build 下拉菜单中的 Build fl.exe 选项，连接程序，如图 2.18 所示。

(8) 再选择 Build 下拉菜单中的 Execute fl.exe 选项，运行程序，如图 2.19 所示。

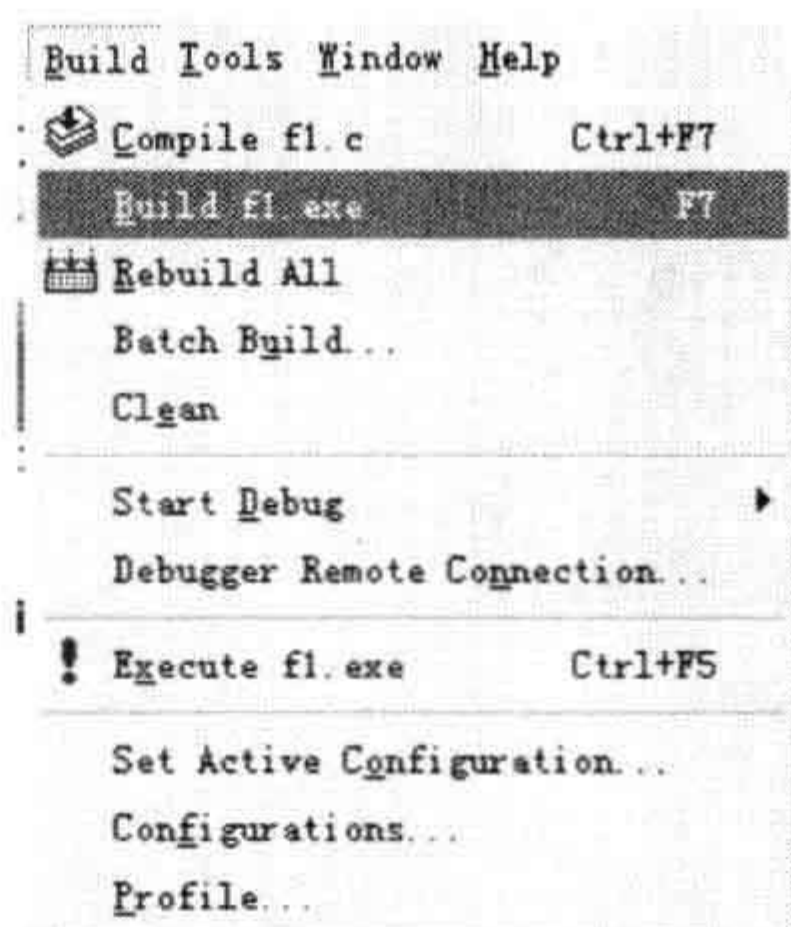


图 2.18 连接

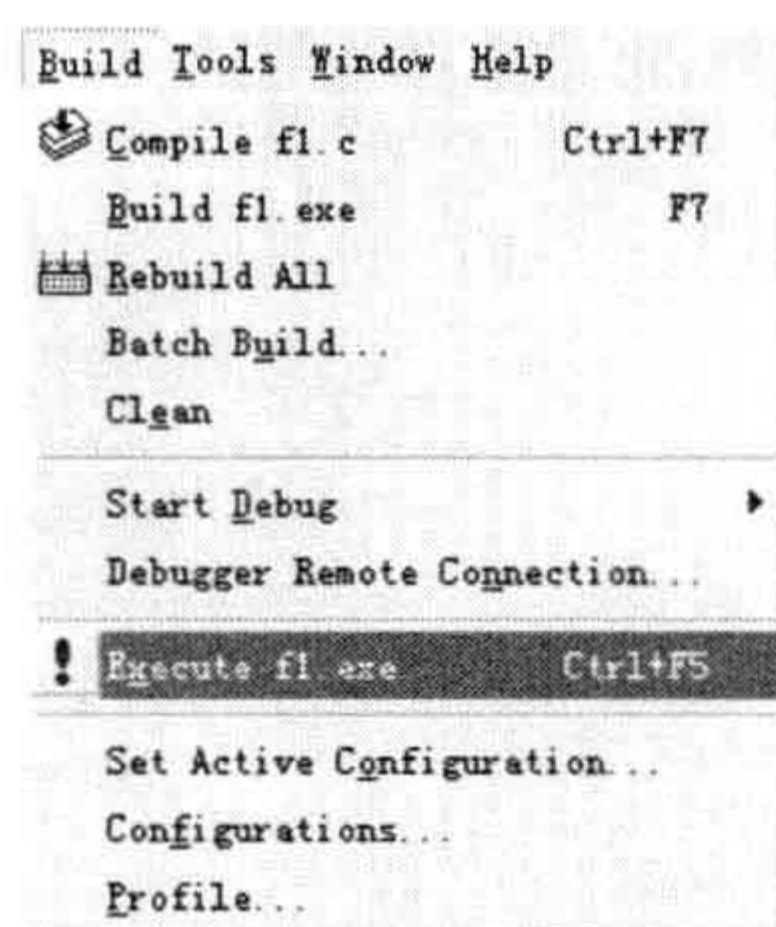



图 2.19 运行

(9) 程序运行界面如图 2.20 所示。按任意键后退出运行界面。

(10) 如果对源程序进行了修改，则需要保存文件。可以通过单击  按钮，或者选择 File 下拉菜单中的 Save 选项，如图 2.21 所示。

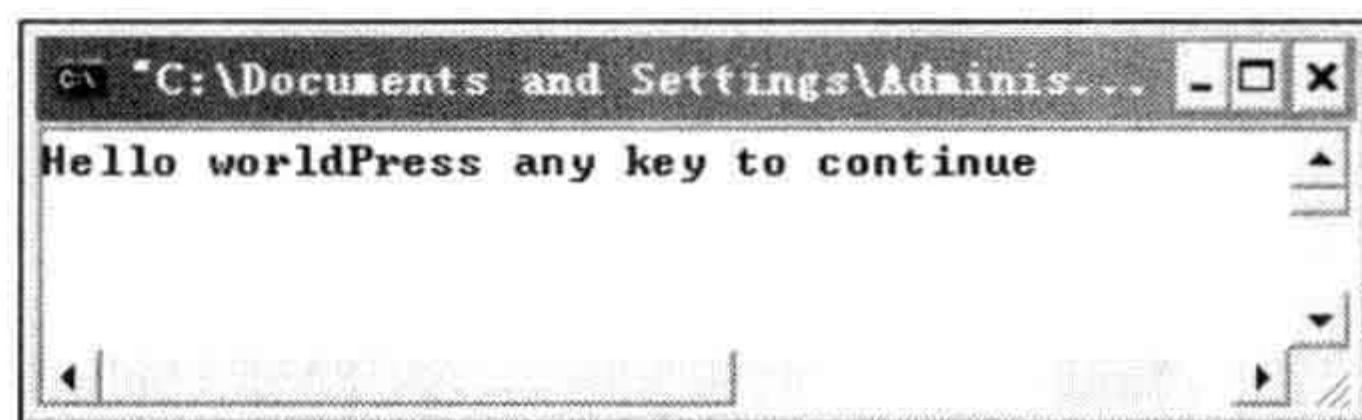


图 2.20 运行结果

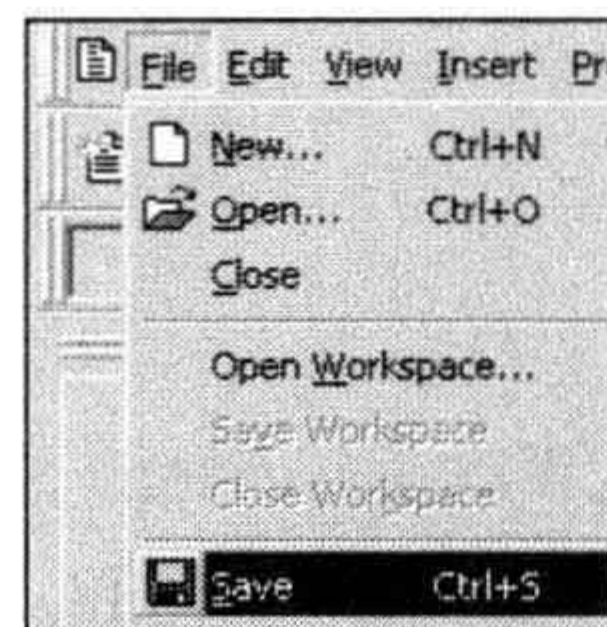



图 2.21 保存

(11) 退出 Visual C++ 6.0，可以通过单击界面右上角的  按钮，也可以选择 File 下拉菜单 Exit 选项，如图 2.22 所示。

(12) 如果有一个已编译好的 C 源文件，需要使用 Visual C++ 6.0 将其打开进行修改，则可以通过选择 File 下拉菜单中的 Open 选项来实现，如图 2.23 所示。





Note

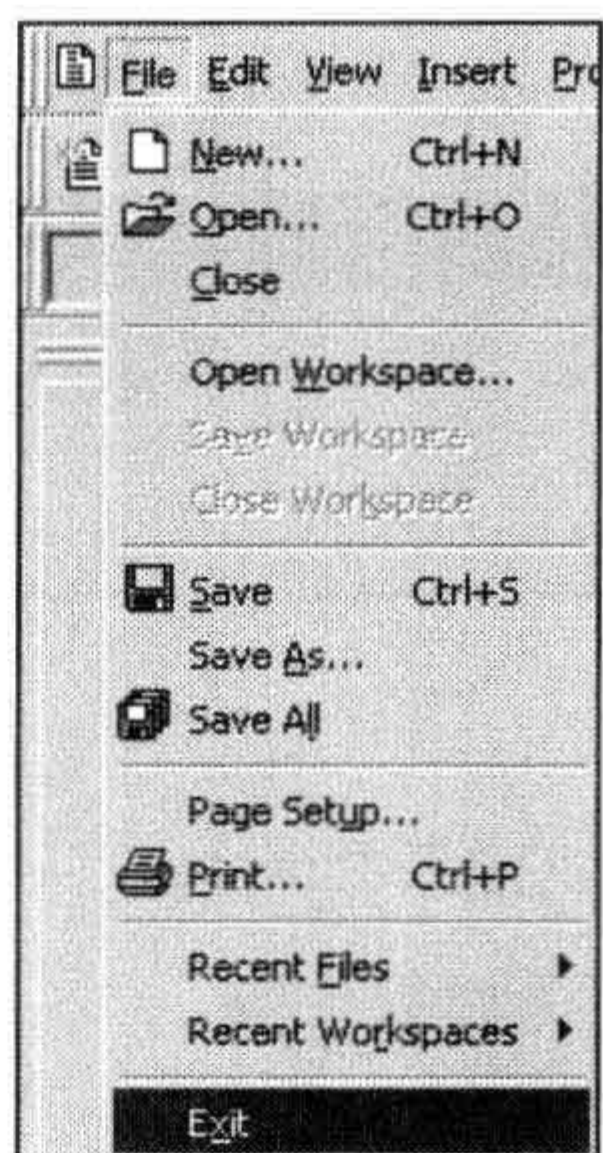


图 2.22 退出

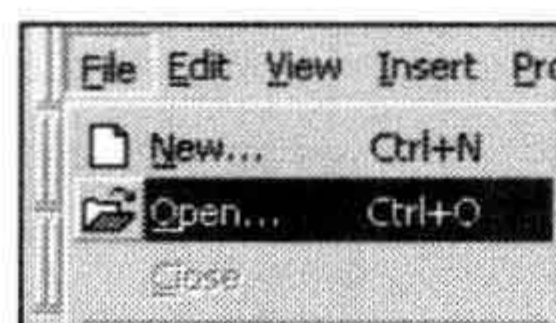


图 2.23 打开文件

(13) 此时会弹出一个对话框，选择要打开的文件，单击“打开”按钮即可，如图 2.24 所示。



图 2.24 选择要打开的文件

专家点评

Visual C++ 6.0 与 Turbo C 2.0 编译器执行的标准不同，所包含的库文件也有所不同，所以在使用的时候要注意它们的区别，以免产生错误。

问题 15 如何在 Visual C++ 2008 中运行一个 C 程序？

问题阐述

Visual C++ 2008 是 Visual Studio 2008 的成员之一。与 Visual Studio 6.0 相比，它具有更强大的功能和人性化的界面。那么如何在 Visual C++ 2008 中运行一个 C 程序呢？

专家解答

具体实现步骤如下。

(1) 在安装了 Visual C++ 2008 之后，通过开始菜单启动 Visual C++ 2008，如图 2.25 所示。

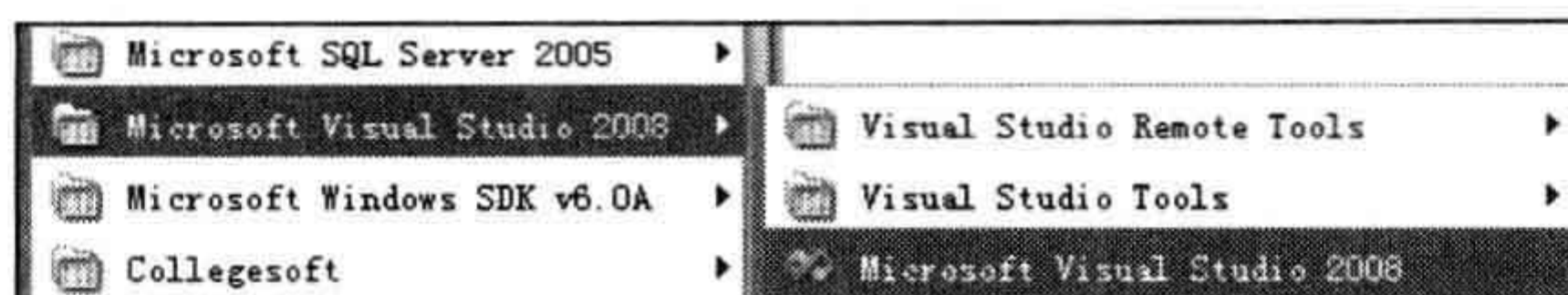


图 2.25 启动 Visual Studio 2008

(2) 打开主菜单“文件”，选择下拉菜单中的“新建”选项，再选中弹出菜单中的“项目”选项，如图 2.26 所示。

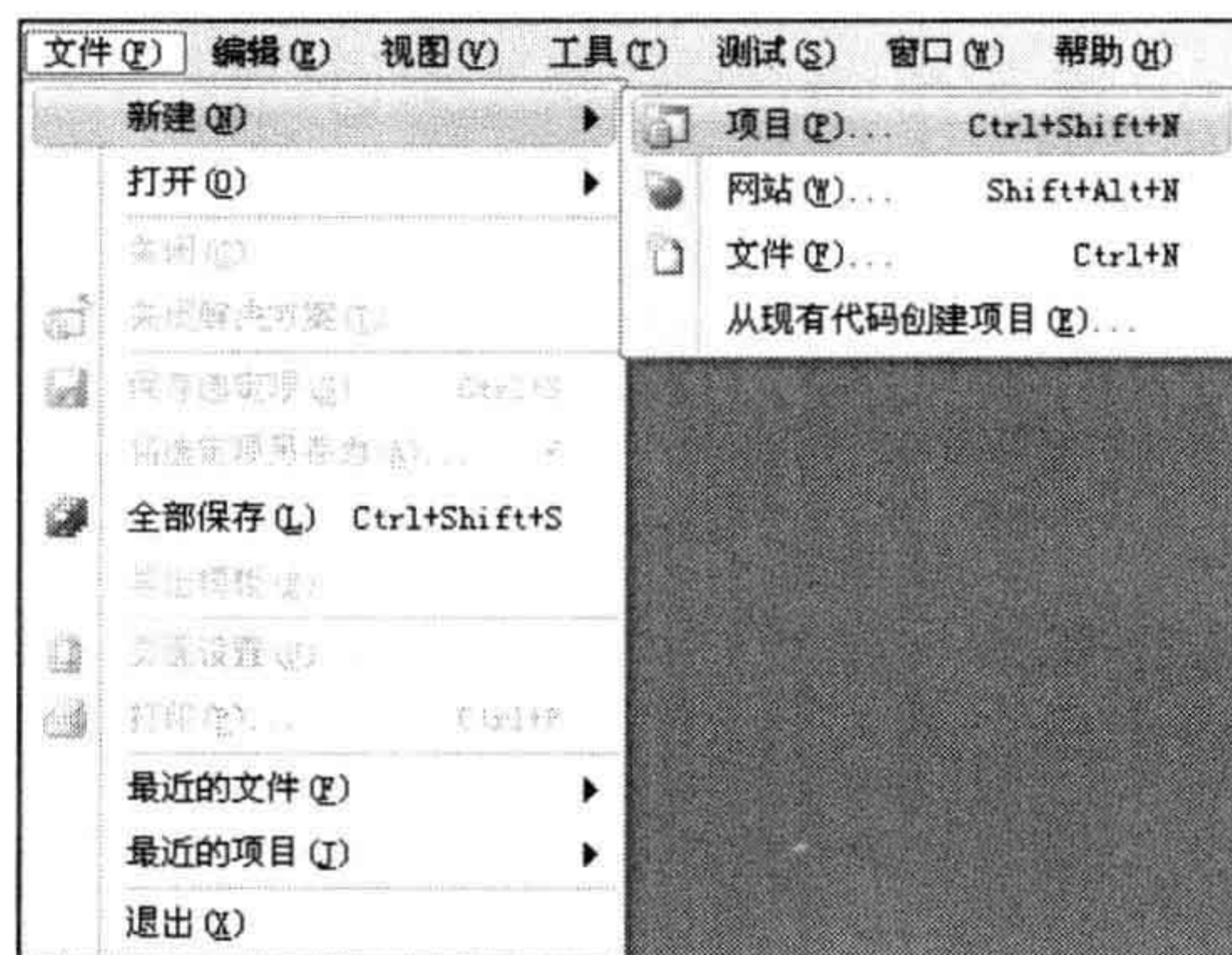


图 2.26 新建项目

此时进入“新建项目”页面，如图 2.27 所示。

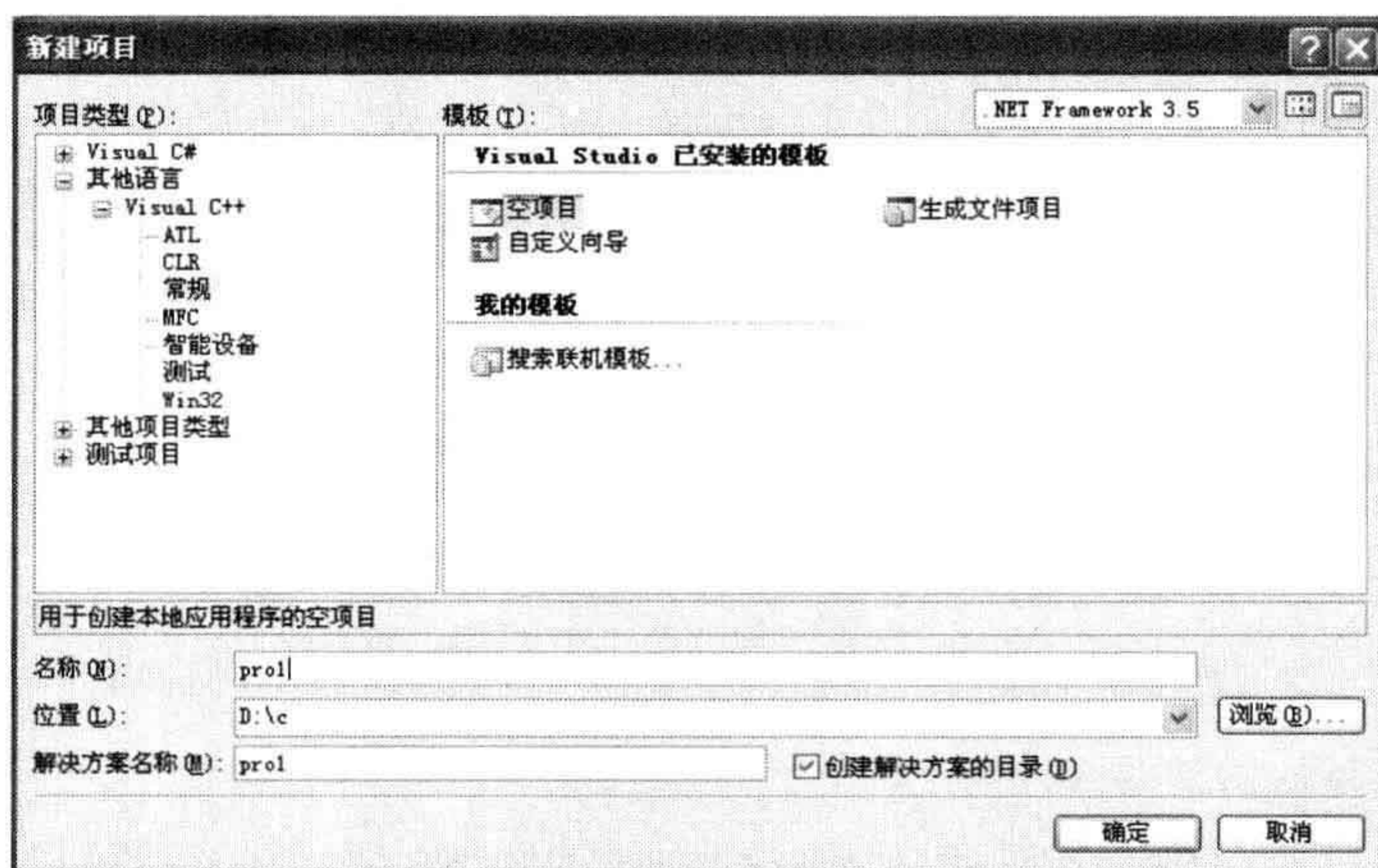


图 2.27 指定空项目名称及位置

(3) 在“新建项目”界面左侧的“项目类型”一栏中选择“常规”选项，再从“模板”一栏中选择“空项目”。在页面下方“名称”文本框中填写要创建的工程名称。在“位置”文本框中填写创建的工程所存放的地址，可通过后面的“浏览”按钮执行设定。填完后，单击下面的“确定”按钮，便完成了一个空的工程的建立。

(4) 选择“视图”下拉菜单中的“解决方案管理器”，如图 2.28 所示。



Note



Note



图 2.28 选择解决方案资源管理器

此时可以看见“解决方案资源管理器”一栏，里面描述了这个工程的结构，工程中包含了三个文件夹，即“头文件”、“源文件”和“资源文件”，如图 2.29 所示。

(5) 工程创建完成后便要创建 C 程序，单击“项目”下拉菜单中的“添加新项”，如图 2.30 所示。

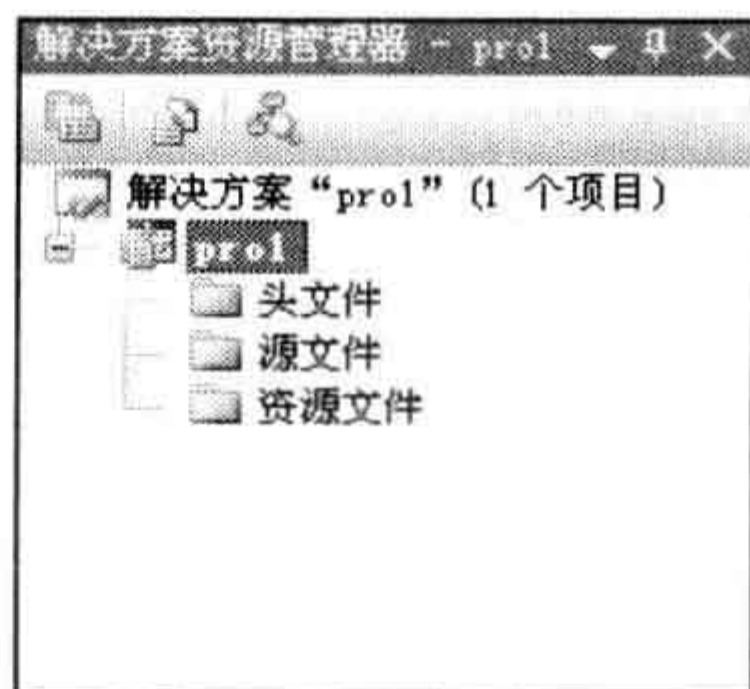


图 2.29 解决方案资源管理器



图 2.30 选择添加新项

可以得到“添加新项”页，如图 2.31 所示。

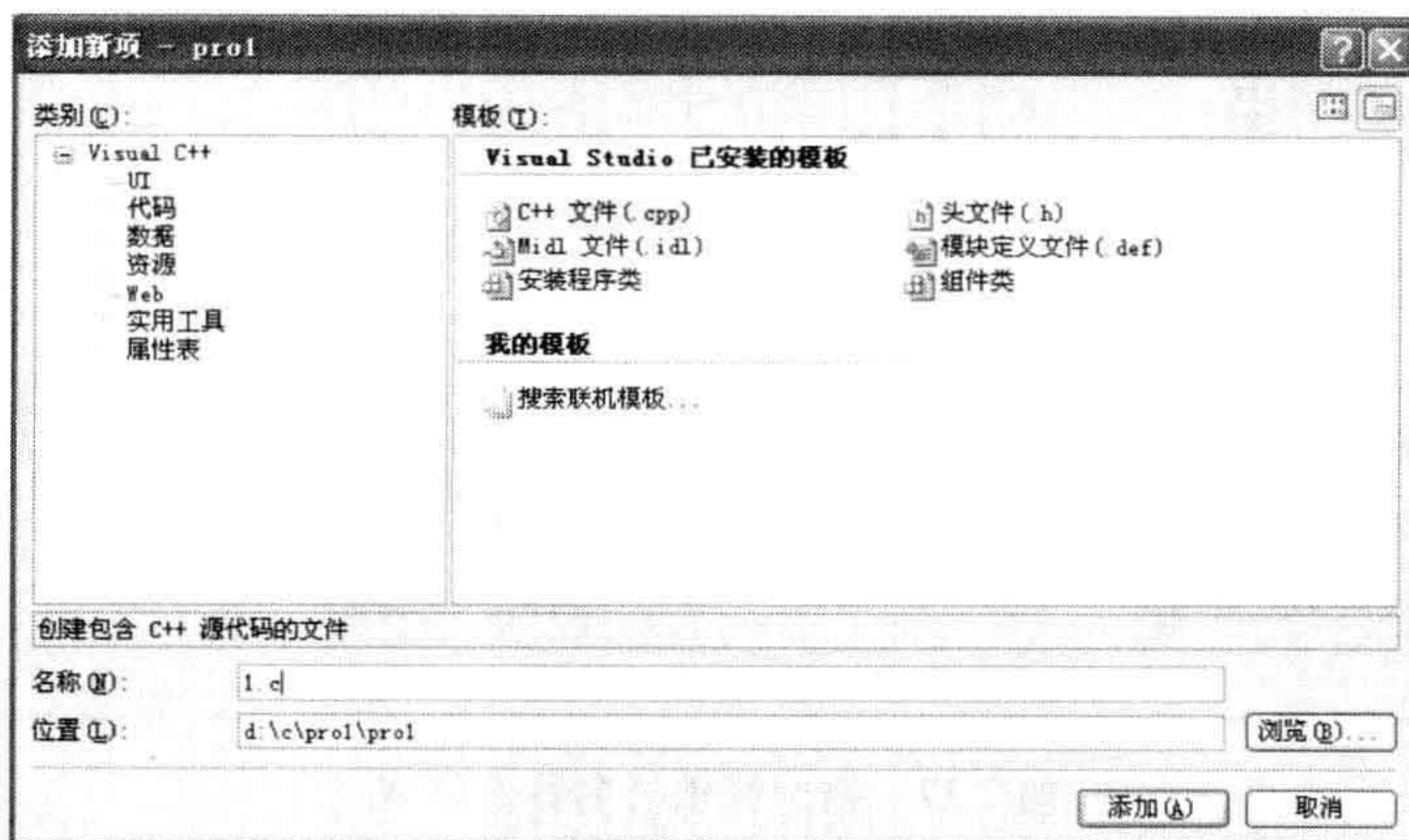


图 2.31 添加新项界面

(6) 在“添加新项”页左侧的“类别”一栏中选择“代码”，再从“模板”一栏中选择“C++文件 (.cpp)”，在下面的“名称”文本框中输入完整的文件名（包括 C 源文件的扩展名.c），“位置”一栏会自动设定，一般情况下不用做更改，单击“添加”即可。

(7) 在完成上述操作后，可建立一个只含有一个.c 文件的工程，如图 2.32 所示。

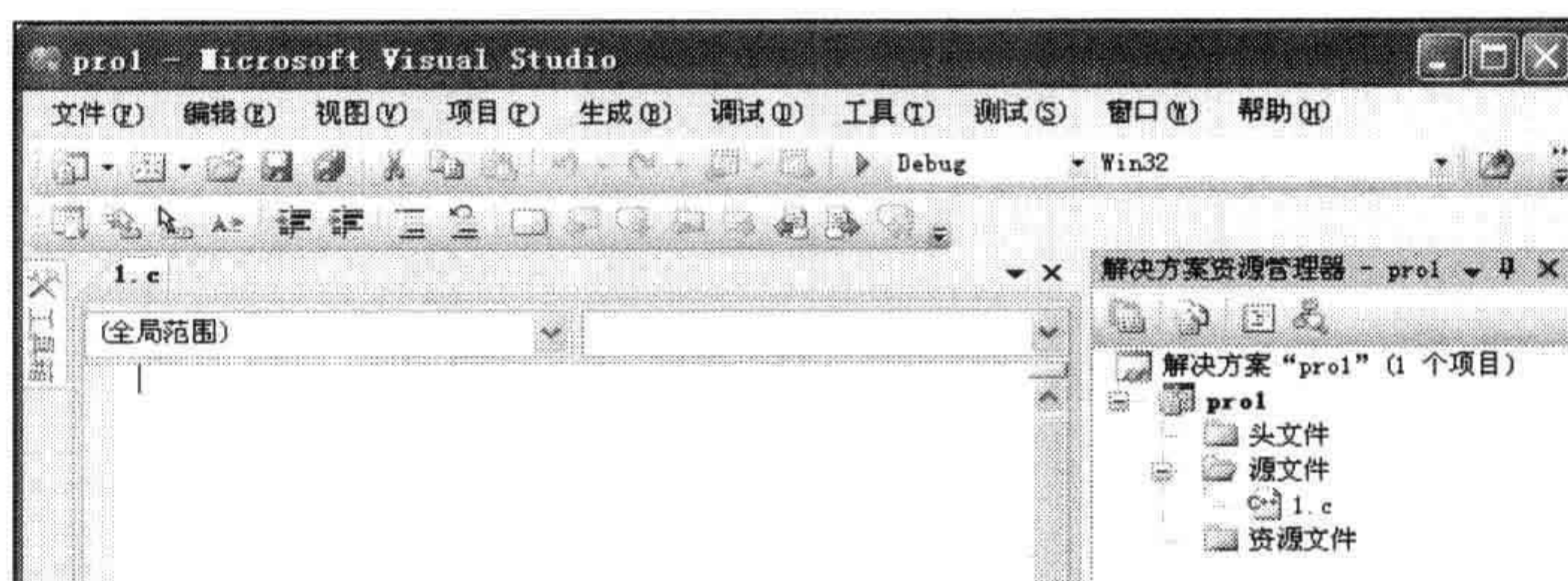


图 2.32 编辑程序界面

在此界面中便可以编写简单的 C 程序，当编写好一个简单的 C 程序之后，完成编译连接并选择“调试”下拉菜单中的“开始执行（不调试）”，就可以运行 C 程序，如图 2.33 所示。

输出 hello mingri 字符串的运行结果如图 2.34 所示。

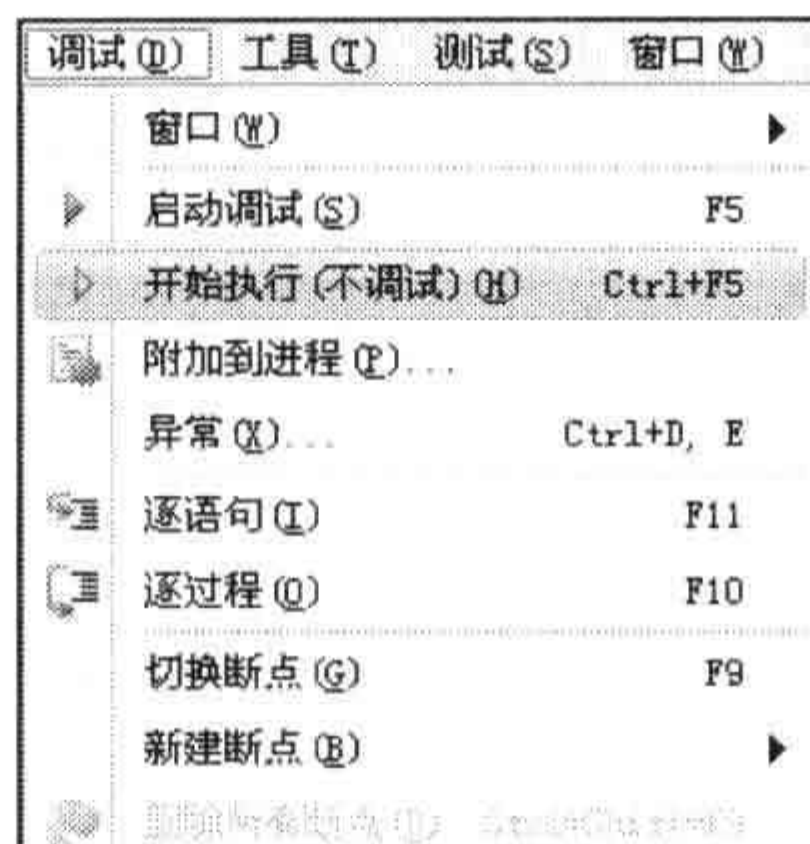


图 2.33 执行程序

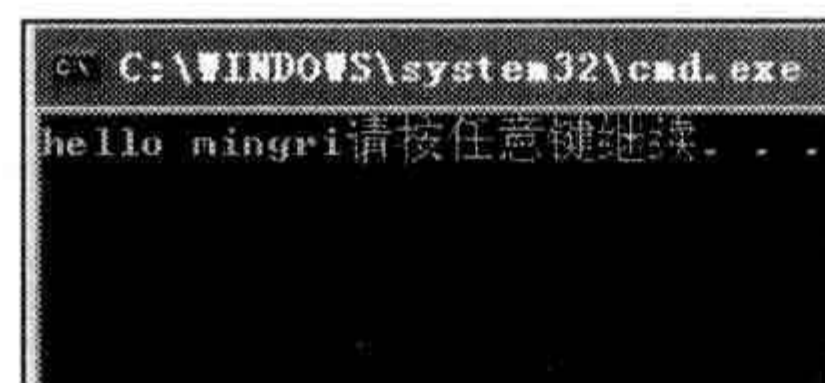


图 2.34 程序运行界面

(8) 当文件修改后需要保存时，可以通过单击“保存”按钮，或者选择“文件”下拉菜单中的“保存”，如图 2.35 所示。

(9) 要退出该界面时，可以通过单击 ，或者选择“文件”下拉菜单中的“退出”，如图 2.36 所示。

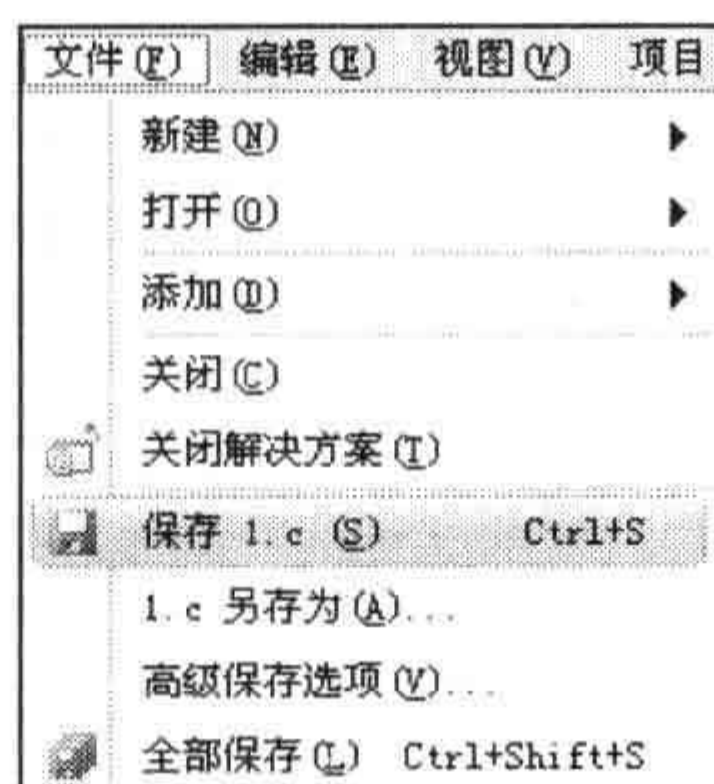


图 2.35 保存文件

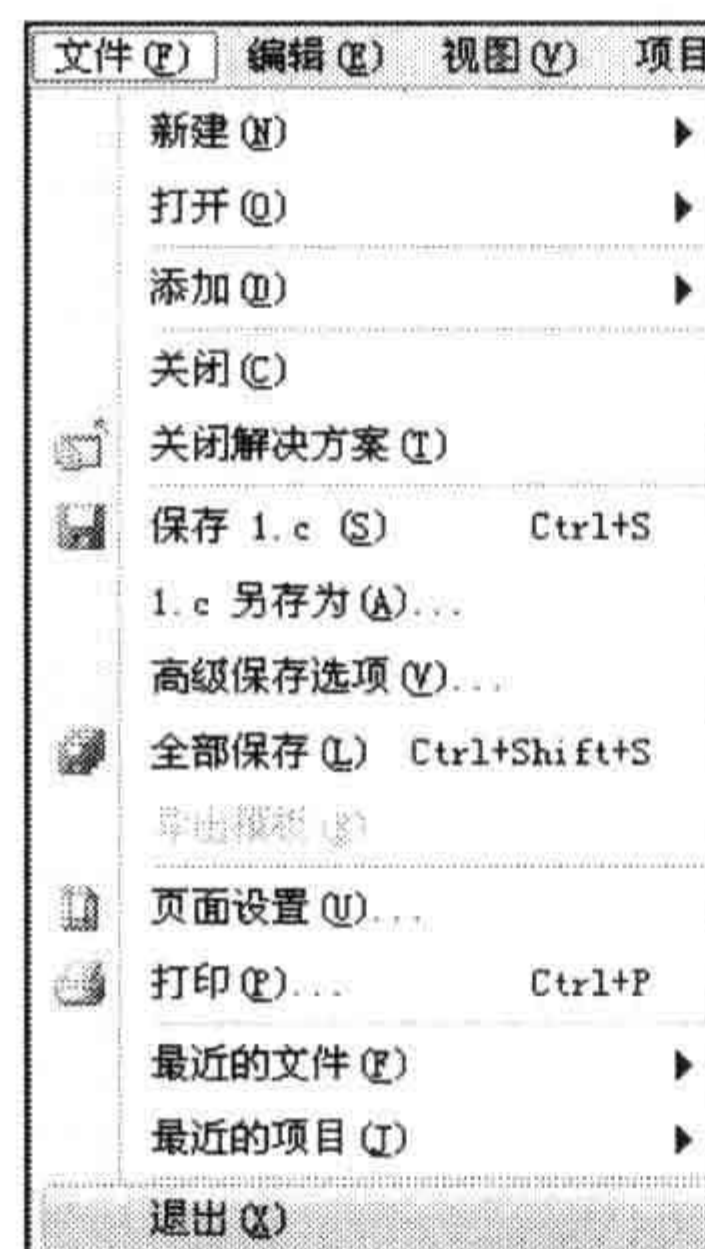


图 2.36 退出

专家点评

以上内容介绍了在三种不同环境下如何运行 C 源文件。对于初学 C 语言的人，建议使





用 Turbo C 2.0。对 C 语言有了深入地了解后，可以使用其他编译工具，包括本书中没有介绍的一些在 Linux 下的编辑器。



Note

问题 16 如何提高程序的可读性？

问题阐述

可读性是评价程序质量的一个重要标准，直接影响到程序的修改和后期维护，那么如何提高程序的可读性呢？

专家解答

提高程序可读性可以从以下几方面来进行。

(1) C 程序整体由函数构成的。程序中，`main()`就是其中的主函数。在程序中是可以定义其他函数的，在这些定义函数中进行特殊的操作，使函数完成特定的功能。将所有的执行代码全部放入 `main()` 函数，虽然程序也是可行的，但是如果将其分成一块一块的，每一块使用一个函数进行表示，那么整个程序看起来就具有结构性，并且易于观察和修改。

(2) 函数体的内容在“{}”中。每一个函数都要执行特定的功能，那么怎么能看出一个函数的具体操作的范围呢？答案就是找寻“{”和“}”这两个花括号。C 语言使用一对花括号来表示程序的结构层次，需要注意的就是左右花括号要对应使用。函数的具体操作范围如图 2.37 所示。



图 2.37 函数的具体操作范围

说明：

在编写程序时，为了防止对应花括号的遗落，每一次都先将两个对应的花括号写出来，然后再向括号中添加代码。

(3) 英文字符大小通用。在程序中，可以使用英文的大写字母，也可以使用英文的小写字母。一般情况下使用小写字母多一些，因为小写字母易于观察。但是，在定义常量时常常使用大写字母，在定义函数时有时也会将第一个字母大写。

(4) 空格、空行的使用。空行的作用就是为了增加程序的可读性。使用适量的空格和空行，可以使程序代码位置安排更合理、美观。但是变量名、函数名和 C 语言保留字中间不能加入空格。除此之外的空格和空行可以任意根据程序可读性和美观的需要进行设



置，C语言编译系统是不会理会这些空格和空行的。

例如，下面代码的书写就非常不利于观察。

```
int max(int a,int b)          /*定义取两数最大者函数*/{
int c;                        /*定义存放最大数的变量*/
c=a>b?a:b;                    /*将两数中较大的赋给c*/
return c;
/*返回最大的数c*/
}
```

**Note**

但是，如果将其中的执行语句在函数中进行一下缩进，使函数体内代码开头与函数头的代码不在一行，这样就会有层次感，例如下面的代码。

```
int max(int a,int b)          /*定义取两数最大者函数*/
{
    int c;                    /*定义存放最大数的变量*/
    c=a>b?a:b;                /*将两数中较大的赋给c*/
    return c;                 /*返回最大的数c*/
}
```

专家点评

不要认为代码实现了功能就是成功的程序代码。一个高质量的程序代码要求具有良好的编写风格，但是也不能一味注重风格，而忽略了代码实现的这个最主要的目的。

问题 17 什么是关键字？C语言的关键字有哪些？

问题阐述

在C语言中常常提到关键字，那么什么是关键字？C语言又有哪些关键字呢？

专家解答

关键字是由C语言规定的具有特定意义的字符串，通常也称为保留字。用户定义的标识符不应与关键字相同。C语言的关键字分为以下几类：

(1) 类型说明符。用于定义、说明变量、函数或其他数据结构的类型。如前面问题中用到的 `int` 等。

(2) 语句定义符。用于表示一个语句的功能。如表 2.1 中的 `if` 和 `else` 就是条件语句的语句定义符。

(3) 预处理命令字。用于表示一个预处理命令。

C语言中有 32 个关键字。在今后的学习中，我们将逐渐接触到这些关键字。具体使用方法如表 2.1 所示。



表 2.1 C 语言中的关键字

关 键 字	意 义
int	声明整型变量或函数
char	声明字符型变量或函数
short	声明短整型变量或函数
long	声明长整型变量或函数
float	声明浮点型变量或函数
double	声明双精度变量或函数
enum	声明枚举类型
static	声明静态变量
const	声明只读变量
struct	声明结构体变量或函数
signed	声明有符号类型变量或函数
unsigned	声明无符号类型变量或函数
union	声明联合数据类型
register	声明寄存器变量
typedef	声明自定义类型
void	声明函数无返回值，声明无参数，声明无类型指针
Auto	声明自动变量，默认时编译器一般认为auto（一般不使用）
break	跳出当前循环
case	开关语句分支（事件处理，根据switch的条件）
continue	重新执行循环体
default	default: 开关语句中的“其他”分支
do	循环语句的循环体
else	条件语句否定分支（与 if 连用）
extern	声明变量是在其他文件中声明（也可以看做是引用外部变量）
for	循环语句
goto	无条件跳转到用户定义的标识符
while	循环语句的循环条件
sizeof	计算数据类型长度
switch	在开关语句中使用（事件判断，执行相关case）
return	返回函数值（可以带参数，也可以不带参数）
volatile	说明变量在程序执行中可被隐含地改变
if	条件语句

专家点评

在 C 语言中，关键字是不允许作为普通的标识符出现在程序中的，但是它可以作为宏名，因为所有预处理发生在识别这些关键字之前。



问题 18 什么是标识符？使用标识符的注意事项是什么？

问题阐述

什么是标识符？使用标识符的注意事项有哪些？

专家解答

C语言在程序的运行过程中，为了可以使用变量、常量、函数、数组等，就要为这些形式设定一个名称，而设定的名称就是所谓的标识符。

在国外，很多外国人的名字是将名字放在前面，而将家族的姓氏放在后面。在中国却恰恰相反，先把姓氏放在前面，而将名字放在后面。从中可以看出，名字是可以随便起的，但是也要按照当地的习俗进行更改。在C语言中，设定一个标识符的名称是非常自由的，可以设定自己喜欢、容易理解的名字，但应该在一定条件下自由发挥。下面介绍一下有关设定C语言标识符应该遵守的一些命名规则。

(1) 所有标识符必须由字母或下划线开头，而不能使用数字或者符号作为开头。可以通过下面的一些正确的写法和错误的写法进行一下比较，例如：

<code>int !number;</code>	<code>/*错误，标识符第一个字符不能为符号*/</code>
<code>int 2hao;</code>	<code>/*错误，标识符第一个字符不能为数字*/</code>
<code>int number;</code>	<code>/*正确，标识符第一个字符为字母*/</code>
<code>int _hao;</code>	<code>/*正确，标识符第一个字符为下划线*/</code>

(2) 在设定标识符时，除标识符开头外，其他位置都可以用字母、下划线或数字，例如：

☒ 在标识符中，有下划线的情况。

<code>int good_way;</code>	<code>/*正确，标识符中可以有下划线*/</code>
----------------------------	--------------------------------

☒ 在标识符中，有数字的情况。

<code>int bus7;</code>	<code>/*正确，标识符中可以有数字*/</code>
<code>int car6V;</code>	<code>/*正确*/</code>

注意：

虽然在设定标识符时，数字是不允许放在一个标识符的开头位置的，但是数字可以放在标识符中。一些符号同样是不允许放在一个标识符的开头位置，而且放在标识符中也是不允许的。

例如：

<code>int love!you;</code>	<code>/*错误，符号不允许放在标识符中*/</code>
<code>int love!;</code>	<code>/*错误*/</code>



Note



Note

(3) 英文字母的大小写代表不同的标识符。也就是说，C 语言中是区分大小写字母的。下面列举出一些标识符，如：

<code>int mingri;</code>	<code>/*全部是小写*/</code>
<code>int MINGRI;</code>	<code>/*全部是大写*/</code>
<code>int MingRi;</code>	<code>/*一部分是小写，一部分是大写*/</code>

从这些列举出的标识符可以看出，只要标识符中的字符有一项是不同的，那么代表的就是一个新的名称。

(4) 标识符不能是关键字。关键字是定义一种类型时使用的字符，标识符是不能使用的。例如，定义第一个整型时，会使用 `int` 关键字进行定义，但是定义标识符就不能使用 `int`。如果将其中标识符的字母改写成大写字母，就可以通过编译。

<code>int int;</code>	<code>/*错误！*/</code>
<code>int Int;</code>	<code>/*正确，改变标识符中的字母为大写*/</code>

(5) 标识符的命名最好具有相关的含义。将标识符设定成有一定含义的名称，这样可以方便程序的编写，并且以后想再进行回顾时，或者他人想进行阅读时，具有含义的标识符使程序便于观察、阅读。例如，我们用两种方式，分别定义一个长方体的长、宽和高对比如下。

<code>int a;</code>	<code>/*代表长度*/</code>
<code>int b;</code>	<code>/*代表宽度*/</code>
<code>int c;</code>	<code>/*代表高度*/</code>
<code>int iLong;</code>	
<code>int iWidth;</code>	
<code>int iHeight;</code>	

从上面列举的标识符可以看出，标识符的设定如果不具有一定的含义，那么没有后面的注释是很难理解这些标识符要代表的意义是什么。若标识符的设定具有其功能含义，那么通过直观的查看就可以了解到其具体的功能。

专家点评

ANSI 标准规定，标识符可以为任意长度，但外部名必须至少能由前 8 个字符区分。这是因为某些编译程序（如 IBM PC 的 MS C）仅能识别前 8 个字符。

问题 19 void 关键字都有哪些用途？

问题阐述

`void` 关键字在 C 语言中有多重意义，那么它都有哪些用途呢？



专家解答

`void` 的字面意思是“空类型”。那么就会有人不解了，空类型又有什么意义呢？下面就来详细介绍 `void` 定义的类型的作用。

(1) `void` 类型的变量。`void` 类型的变量没有任何的意义，因此在编程的时候也不会有人定义一个 `void` 类型的变量。如下面的代码。

```
void m;
```

在一些编译器上，这样的代码可能会产生一个编译错误，即使没有错误，这样的代码也没有任何的实际意义。但是可以将一个指针变量定义为 `void` 类型，即定义一个 `void*` 类型的变量。`void*` 为空类型指针，它可以指向任何类型的变量。空类型指针在后面章节中将有详细介绍，这里不再赘述。

(2) `void` 修饰函数返回值。如果函数没有返回值，可以将其声明为 `void` 类型。在 C 语言中，不加返回类型限定的函数，编译器一般默认其返回类型为整型。而许多人认为，不加返回类型限定的函数其返回类型为 `void`，甚至很多人认为 `main()` 函数是无返回值或者为 `void` 类型的，这是不对的。所以，如果定义的函数没有返回值，一定要声明为 `void` 类型，这样也能增加程序的可读性，更是良好编程风格的体现。

(3) `void` 修饰函数参数。与修饰函数返回值一样的，如果函数没有参数，应该使用 `void` 进行修饰，以增强程序的可读性。

专家点评

`void` 体现了一种抽象，它不能代表一个真正的变量。如果理解了面向对象中的“抽象基类”，`void` 类型就好理解了。因为变量是一个真实的存在，定义的时候需要为其分配内存，所以不能定义为空类型。

问题 20 什么是匈牙利命名约定？它是否是好的约定？

问题阐述

经常听说匈牙利命名约定，它是怎样的约定？是否是比较好的约定？

专家解答

匈牙利命名约定由 Charles Simonyi 发明。他把变量的类型和用途等信息编写在变量名中。在某些团队里，它被视为优秀的命名约定。而在另一些地方，却被严厉地批评。它的优势在于变量名显示了类型和用途，它的主要缺点在于将类型信息放在变量名中，这显得很冗余。



Note



专家点评



Note

编程风格可以认为是大家约定俗成的规则。世界上没有最好的编程风格，它需要根据需求而定。团队开发讲究风格一致，这样才能共同开发出好的程序。

第 3 章

算法入门

- ▶▶ 为什么说算法是程序设计的灵魂?
- ▶▶ 算法的特性有哪些?
- ▶▶ 如何评价一个算法的好坏?
- ▶▶ 算法的表示方法都有哪些?
- ▶▶ 算法的基本结构是什么?
- ▶▶ 算法有哪几类?
- ▶▶ 算法的效率度量方法有哪些?
- ▶▶ 什么是算法的时间复杂度?
- ▶▶ 什么是算法的空间复杂度?
- ▶▶ 什么是分治法算法思想?



问题 21 为什么说算法是程序设计的灵魂?



问题阐述

算法对于程序设计来说十分重要,被称为程序设计的灵魂,那么算法为什么被称为程序设计的灵魂呢?

专家解答

很多人认为算法只存在于那些数学家或计算机专业人士的脑海中,其实不然,算法无处不在,只是由于它不是看得见、摸得着的具体物体,所以人们常常忽略它的存在。

算法其实就是为解决一个问题而采取的方法和步骤。例如,洗脸可以简单分成如下几步。

- (1) 将清水倒入盆中;
- (2) 挤上洗面奶,清洗脸部;
- (3) 用水洗净脸上的洗面奶;
- (4) 用毛巾擦干脸。

以上这四步就称之为解决洗脸这个问题的算法。

著名科学家沃思提出一个公式:

数据结构+算法=程序

在计算机程序设计中,数据结构是操作的对象,算法是对对象进行加工处理,用以得到程序的运行结果,程序中的操作语句实际上就是算法的体现。算法与程序设计和数据结构密切相关,是解决一个问题的完整的步骤描述,是解决问题的策略、规则和方法。

如果将计算机程序比喻成有生命的人,那“数据结构”就是人的躯体,算法是人的灵魂。只有躯体与灵魂的相互结合,才能组成一个完完整整的有生命、有思想的人。因此,算法具有程序的灵魂之说。

专家点评

解决一个问题的算法并不是唯一的,可以有多种方法。而这多种方法中,又有时间和空间效率高低之分,所以在设计算法的时候,要考虑到算法的效率。

问题 22 算法的特性有哪些?

问题阐述

每一个事物都有其组成特性,那么算法都要求有哪些特性呢?



专家解答

算法是解决“做什么”和“怎么做”的问题。解决一个问题可能有多种不同的算法，从效率上考虑，其中最为核心的还是算法的速度。因此，解决问题的步骤需要在有限的时间内完成，并且操作步骤中不可以有歧义性语句，以免后继步骤无法继续进行下去。通过对算法概念的分析，可以总结出一个算法必须满足如下5个特性。

(1) 有穷性。一个算法在执行有限步骤后，在有限时间内能够实现的，就称该算法具有有穷性。

有的算法在理论上满足有穷性，在有限的步骤后能够完成，但是计算机可能实际上会执行一天、一年、十年等等。算法的核心就是速度，那么这个算法也就没有意义了。总而言之，有穷性没有特定的限度，取决于人们的需要。

(2) 确定性。算法中每一个步骤的表述都应该是确定的、没有歧义的语句。在人们的日常生活中，遇到歧义性语句，可以根据常识、语境等理解，然而还有可能理解错误。计算机不比人脑，不会根据算法的意义来揣测每一个步骤的意思，所以算法的每一步都要有确定的含义。

(3) 有零个或多个输入。程序中的算法和数据是相互联系的。算法中，需要输入的是数据的量值。输入可以是多个也可以是零个。其实，零个输入并不是这个算法没有输入，而是这个输入没有直观地显现出来，隐藏在算法本身当中。

(4) 有一个输出或多个输出。输出就是算法实现所得到的结果，是算法经过数据加工处理后得到的结果。有的算法输出的是数值，有的是图形，有的输出并不是那么显而易见。没有输出的算法是没有意义的。

(5) 可行性。算法的可行性就是指每一个步骤都能够有效地执行，并得到确定的结果，而且能够用来方便地解决一类问题。

专家点评

算法的这些特性是一个程序算法所必须要有的性质。在编写算法的时候要注意这些特性，只有满足了这些特性，才能说是一个合格的算法。

问题 23 如何评价一个算法的好坏？

问题阐述

算法的好坏评测也被称为算法的性能分析，那么如何评价一个算法的好坏呢？

专家解答

由于针对一个问题可能会有不同的算法去解决，不同的算法思路不同，有的执行起来



Note



会很慢，效率很低，有的执行起来会很快，效率自然会很高。这样，就出现了算法的“好”与“坏”之分，如何衡量一个算法的好坏，通常要从以下几个方面来分析。

(1) 正确性。算法能满足具体问题的需求，即对任何合法的输入，算法都会得出正确的结果。

(2) 可读性。算法创建后由人来阅读、理解、使用以及修改，所以可读性的好坏直接影响到算法的好坏。如果一个算法涉及的想法很多，人就会糊涂，那么这个算法就不能更好地交流和推广使用，自然对修改、扩展、维护就更不方便；所以要提高算法的可读性，让其简明易懂。

(3) 健壮性。一个程序编译完成后，运行该程序的用户对程序的理解各有不同，并不能保证每一个人都能按照要求进行输入。健壮性就是指对非法输入的抵抗能力，当输入的数据非法时，算法能识别并做出处理，而不会因为输入的错误产生错误动作或造成瘫痪。

(4) 时间复杂度与空间复杂度。时间复杂度简单地说就是算法运行所需要的时间。不同的算法具有不同的时间复杂度，当一个程序较小时我们感觉不到时间复杂度的重要性，当一个程序特别大时便会察觉到时间复杂度实际上是十分重要的。所以如何写出更高速的算法一直是算法不断改进的目标。空间复杂度是指算法运行所需的存储空间的多少，随着计算机硬件的发展，空间复杂度已经显得不再那么重要。

专家点评

时间复杂度和空间复杂度是评测算法好坏的重要性能指标，所以在编写算法的时候要尽量减小时间复杂度和空间复杂度。

问题 24 算法的表示方法都有哪些？

问题阐述

算法设计者必须将自己设计的算法清楚、正确地按步骤记录下来，这个过程就叫描述算法。表示一个算法，可以用不同的方法。那么算法的表示方法都有哪些呢？

专家解答

一个算法有多种表述方式，常见的有自然语言、流程图、N-S 图、伪代码、计算机语言等。下面分别进行介绍。

1. 自然语言

所谓自然语言，就是日常生活中的语言。它可以是汉语、英语、日语等，一般用于描述一些简单的问题、步骤，可以使算法通俗、简单易懂。下面通过具体实例来介绍自然语言。





例如, 任意输入三个数, 求这三个数中的最大数。

第一步: 定义四个变量, 分别为 x 、 y 、 z 以及 \max 。

第二步: 输入大小不同的三个数, 分别赋给 x 、 y 、 z 。

第三步: 判断 x 是否大于 y , 如果大于, 则将 x 的值赋给 \max , 否则将 y 的值赋给 \max 。

第四步: 判断 \max 是否大于 z , 如果大于, 则执行步骤五, 否则将 z 的值赋给 \max 。

第五步: 将 \max 的值输出。

自然语言最大的优点就是容易理解, 适用于比较简单的问题。对于比较复杂的问题或者在描述包括分支或循环的算法时一般会很冗长, 所以不用自然语言描述、表示算法, 避免出现二义性。

2. 流程图

流程图是一种传统的算法表示法, 它用一些图框来代表各种不同性质的操作, 用流程线来指示算法的执行方向。由于它简单直观, 易于理解, 所以应用广泛。常见的流程图符号及流程图的例子如图 3.1 所示。

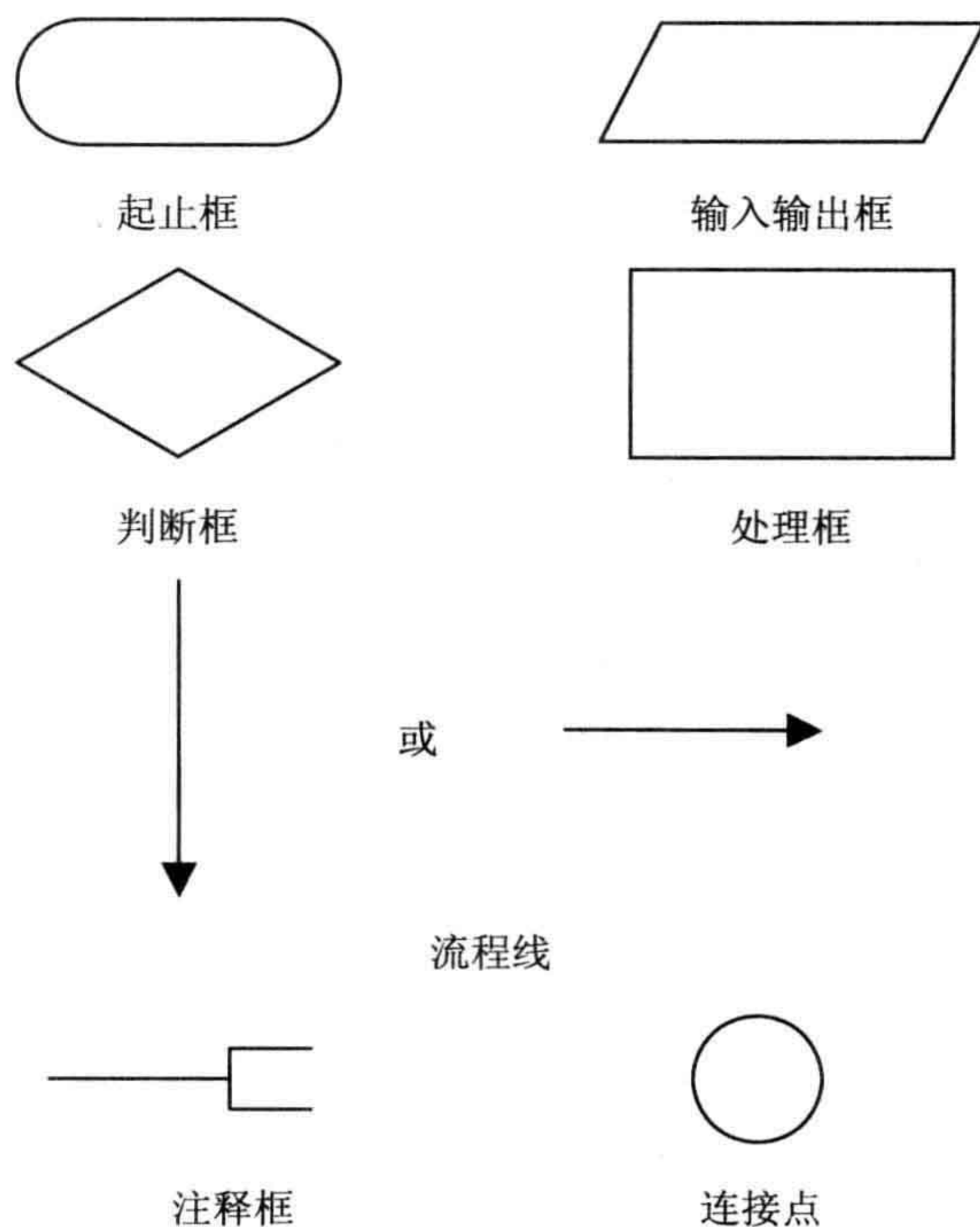


图 3.1 流程图的图元表示方法

其中, 起止框是用来标识算法开始和结束的; 判断框的作用是对一个给定的条件进行判断, 并根据给定的条件是否成立来决定如何执行后面的操作; 连接点是将画在不同地方的流程线连接起来。下面通过几个例子来介绍图框的使用方法。

例如, 求两个整数 a 和 b 的最大公约数。流程图如图 3.2 所示。



Note

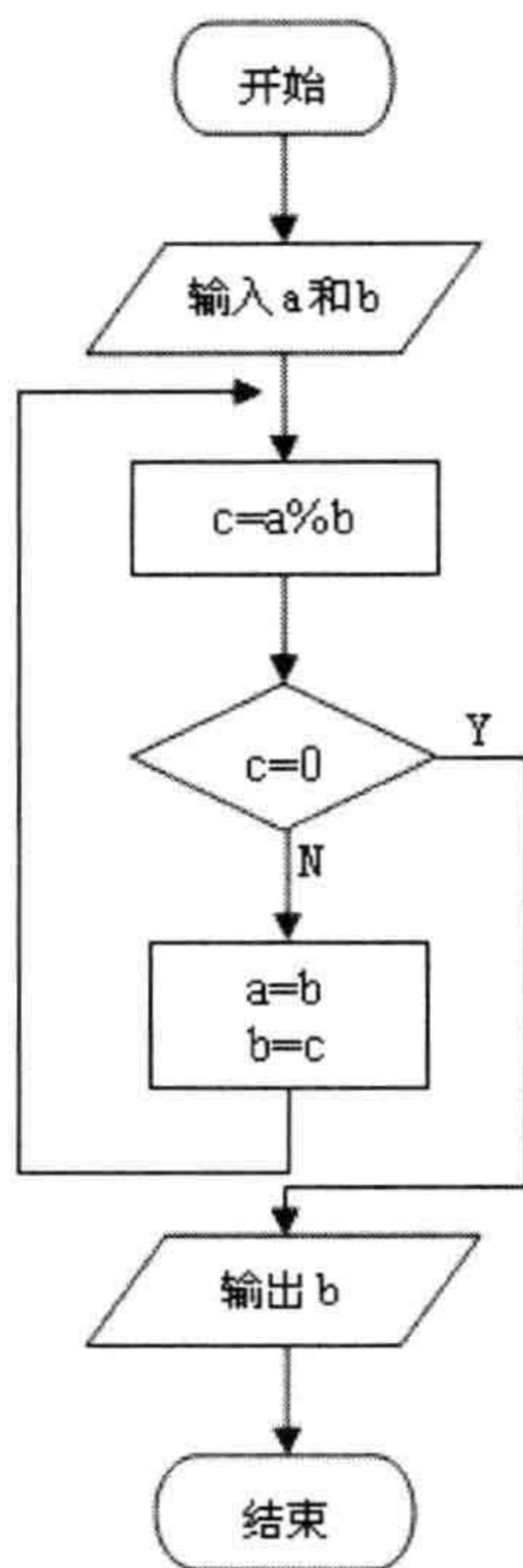


图 3.2 求两个整数 a 和 b 的最大公约数

判断框两侧的“Y”和“N”代表“是”(yes)和“否”(no)。

经过研究发现，任何复杂的算法，都可以由顺序结构、选择结构和循环结构这三种基本结构组成，这三种基本结构之间可以并列、可以相互包含，但不允许交叉，不允许从一个结构直接转到另一个结构的内部去。

3. N-S 图

既然任何算法都是由前面介绍的 3 种结构组成的，那么各基本结构之间的流程线就成了多余的。N-S 流程图（这是由美国人 I.Nassi 和 B.Shneiderman 共同提出的，故以他们名字的首字母命名）去掉了原来的所有流程线，将全部的算法写在一个矩形框内。它也是算法的一种结构化描述方法，同样也有三种基本结构。

(1) 顺序结构的 N-S 流程图，如图 3.3 所示。

(2) 选择结构的 N-S 流程图，如图 3.4 所示。

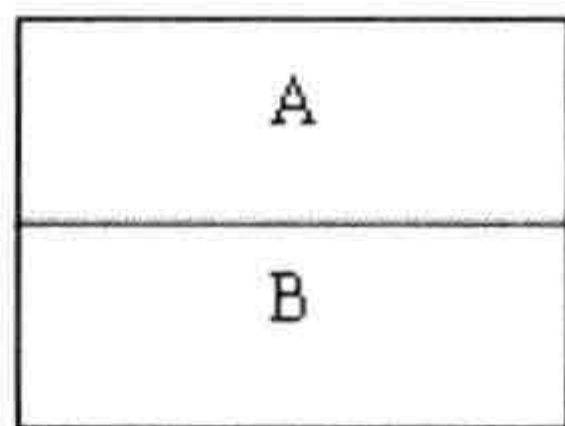


图 3.3 顺序结构

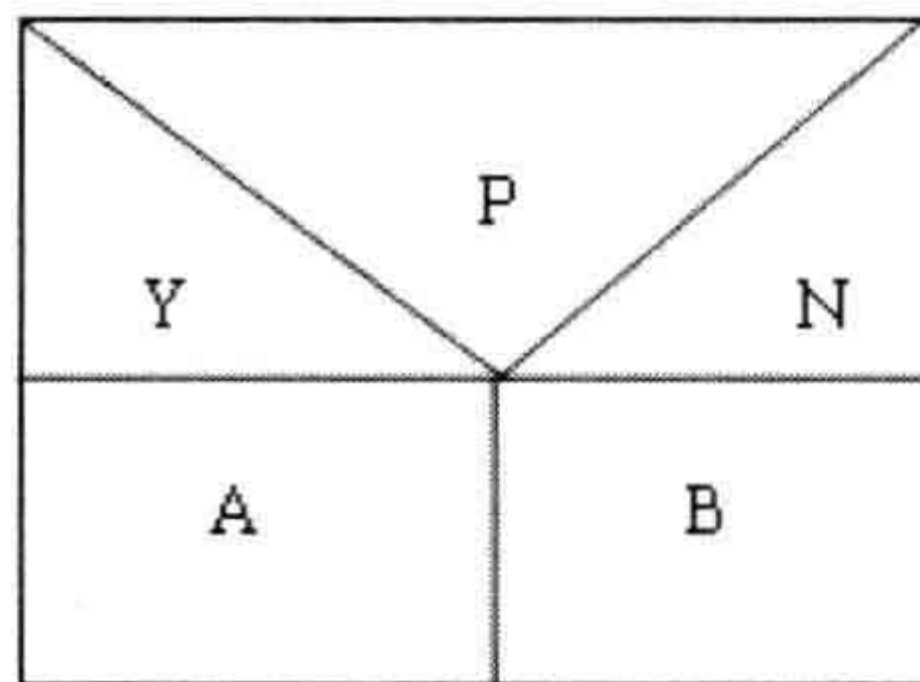


图 3.4 选择结构

例如，输入一个数，判断该数是否是偶数，并给出相应提示。此程序的选择结构的



N-S 流程图如图 3.5 所示。



图 3.5 判断偶数

(3) 循环结构。当型循环的 N-S 流程图，如图 3.6 所示。

例如，程序求 1~100 之间（包括 1 和 100）所有整数之和的当型循环的 N-S 流程图如图 3.7 所示。

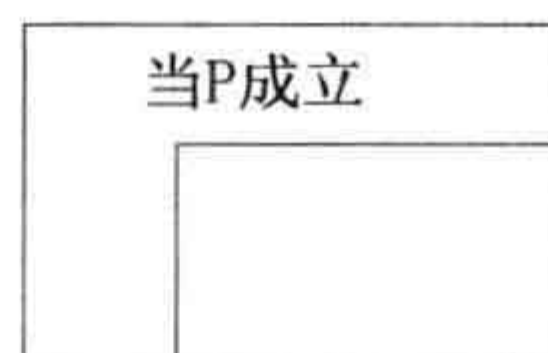


图 3.6 当型循环

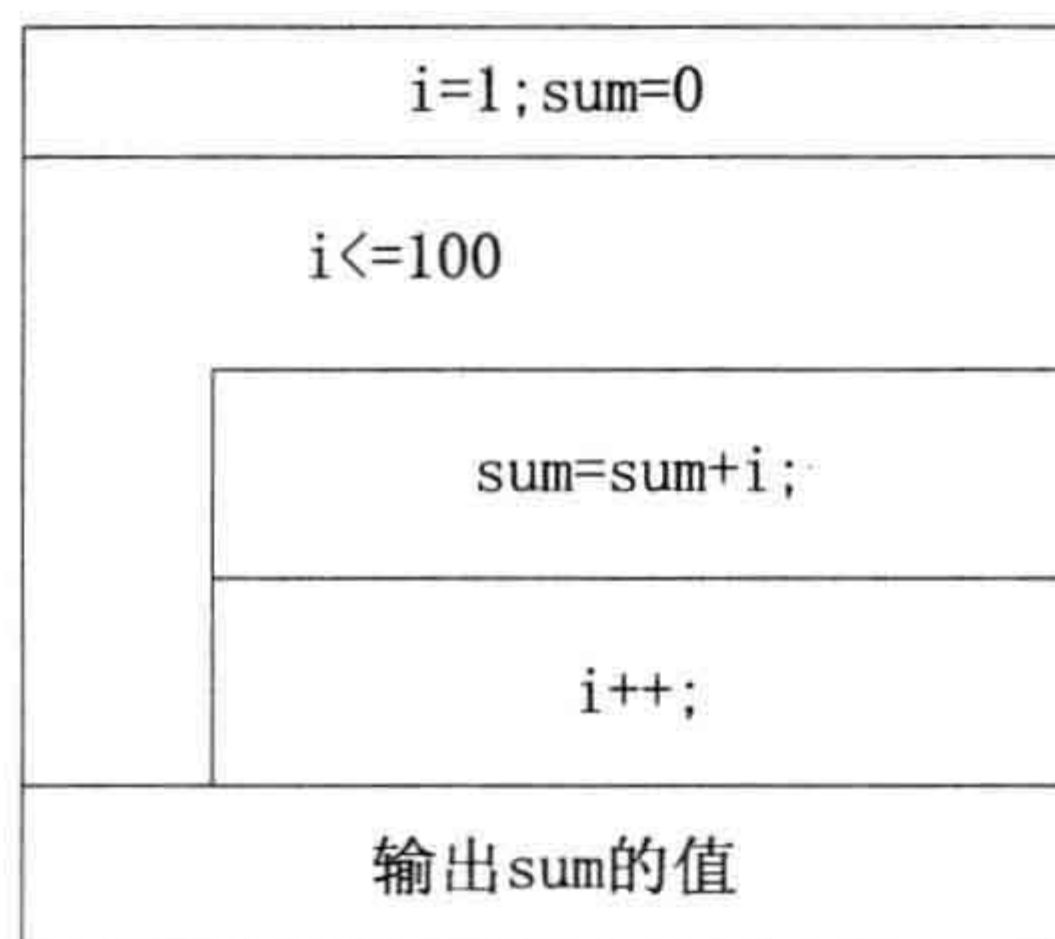


图 3.7 当型循环求和

直到型循环的 N-S 流程图，如图 3.8 所示。

例如，程序求 1~100 之间（包括 1 和 100）所有整数之和的直到型循环的 N-S 流程图如图 3.9 所示。

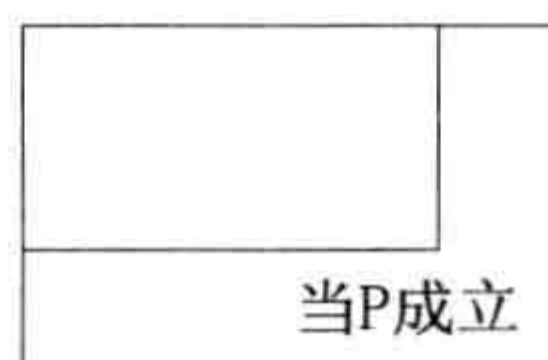


图 3.8 直到型循环

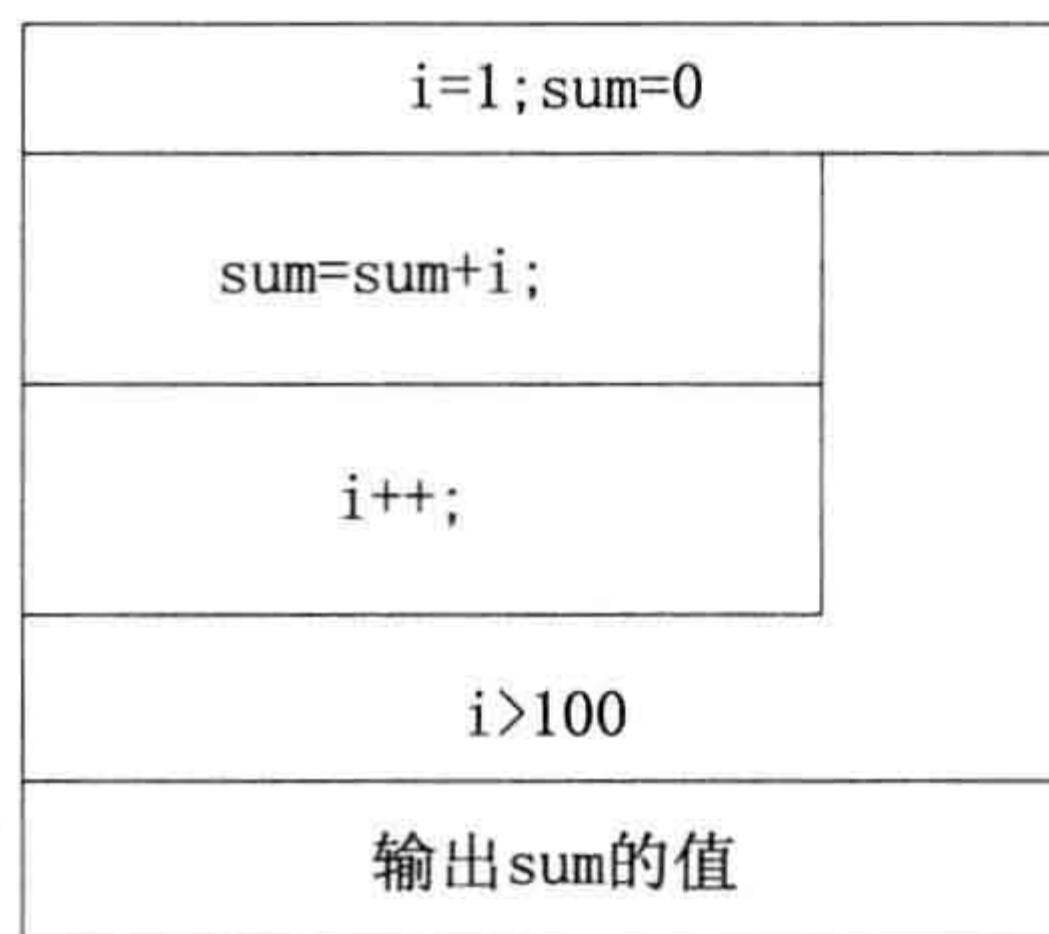


图 3.9 直到型循环求和

4. 伪代码

伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。它采用某一程序设计语言的基本语法，如操作指令，可以结合自然语言来设计。而且，它不用符号，书写方便，没有固定的语法和格式，具有很大的随意性，便于向程序过渡。

下面通过一个例子来介绍如何用伪代码描述算法。





例如，用伪代码描述两个正整数 a 和 b 最大公约数的算法。

```
开始  
c=a%b;  
循环直到 c=0  
    a=b;  
    b=c;  
    c=a%b;  
输出 b;  
结束
```

注意：

伪代码虽然不是一种实际的编程语言，但表达能力上类似编程语言，同时避免了描述技术细节带来的麻烦，所以伪代码更适合描述算法，故被称作“算法语言”或“第一语言”。

专家点评

算法的描述要根据算法的规模和组成特点来选择不同的描述方式。选择合适的描述方式，能够更清晰直接地对算法进行表示。

问题 25 算法的基本结构是什么？

问题阐述

算法有哪几种基本结构？

专家解答

经过研究发现，任何复杂的算法，都可以由顺序结构、选择结构和循环结构这三种基本结构组成，这三种基本结构之间可以并列，可以相互包含，但不允许交叉，不允许从一个结构直接转到另一个结构的内部去。

整个算法都是由三种基本结构组成的，所以只要规定好三种基本结构的流程图的画法，就可以画出任何算法的流程图。

1. 顺序结构

顺序结构是最简单的线性结构，在顺序结构的程序里，各操作是按照它们出现的先后顺序执行的，如图 3.10 所示。

注意：

在执行完 A 框指定的操作后，必须接着执行 B 框所指定的操作。

例如，输入两个数分别赋给变量 i 和 j ，再将这两个数分别输出。



本实例的流程图可以采用顺序结构来实现,如图 3.11 所示。

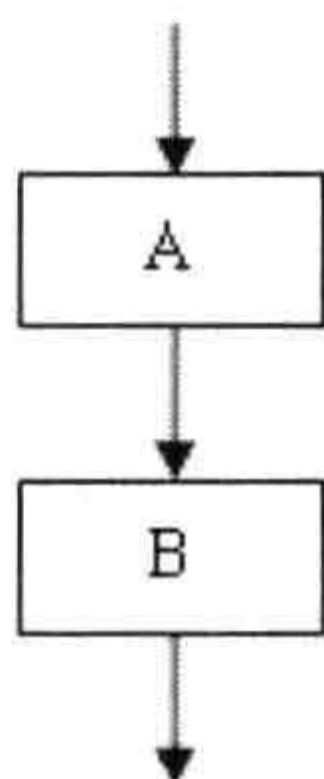


图 3.10 顺序结构

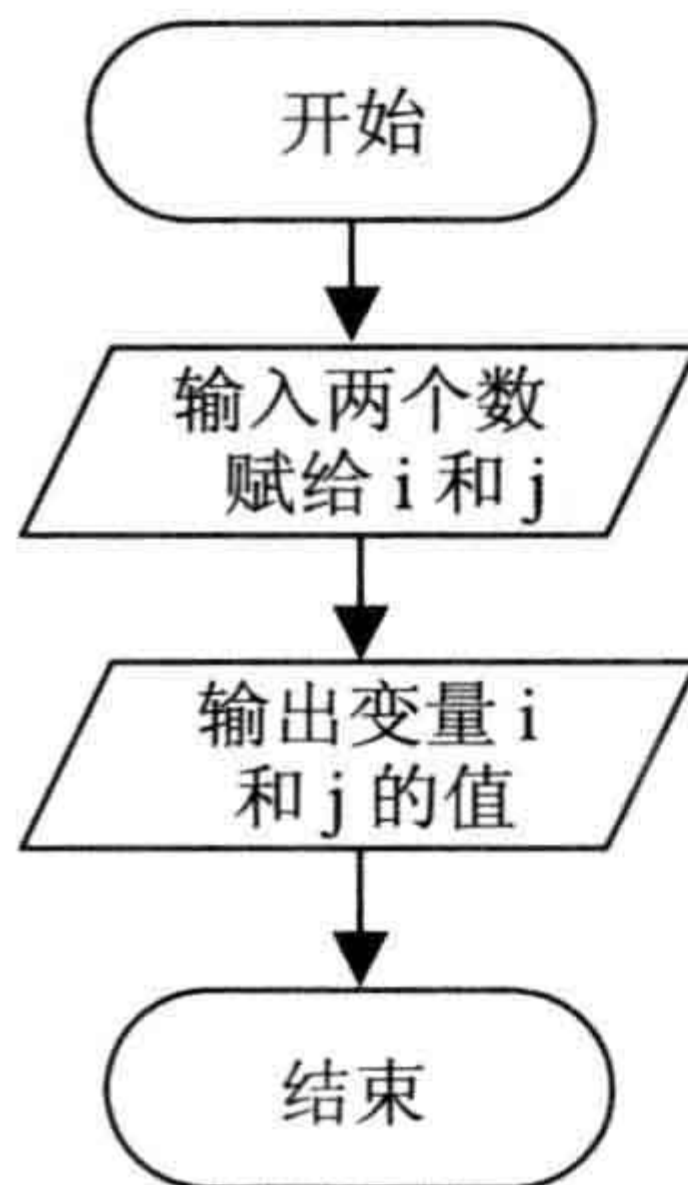


图 3.11 输入两变量的值

2. 选择结构

选择结构也叫分支结构,有两种形式,如图 3.12 和图 3.13 所示。

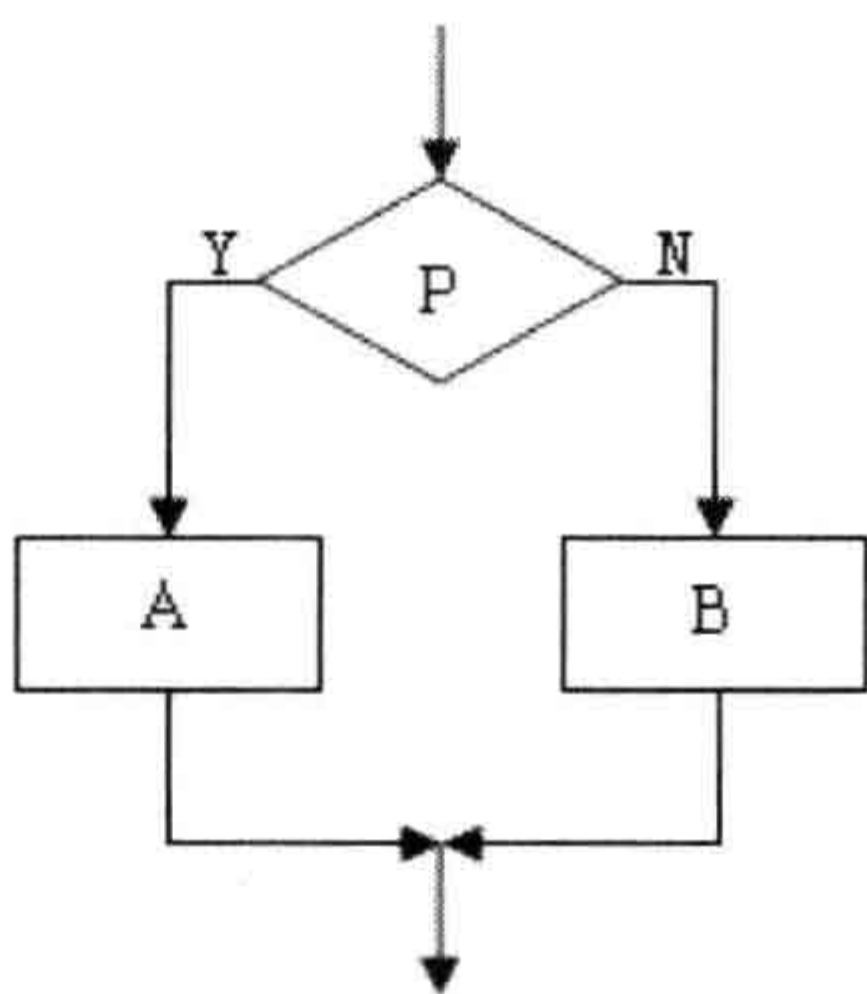


图 3.12 选择结构 1

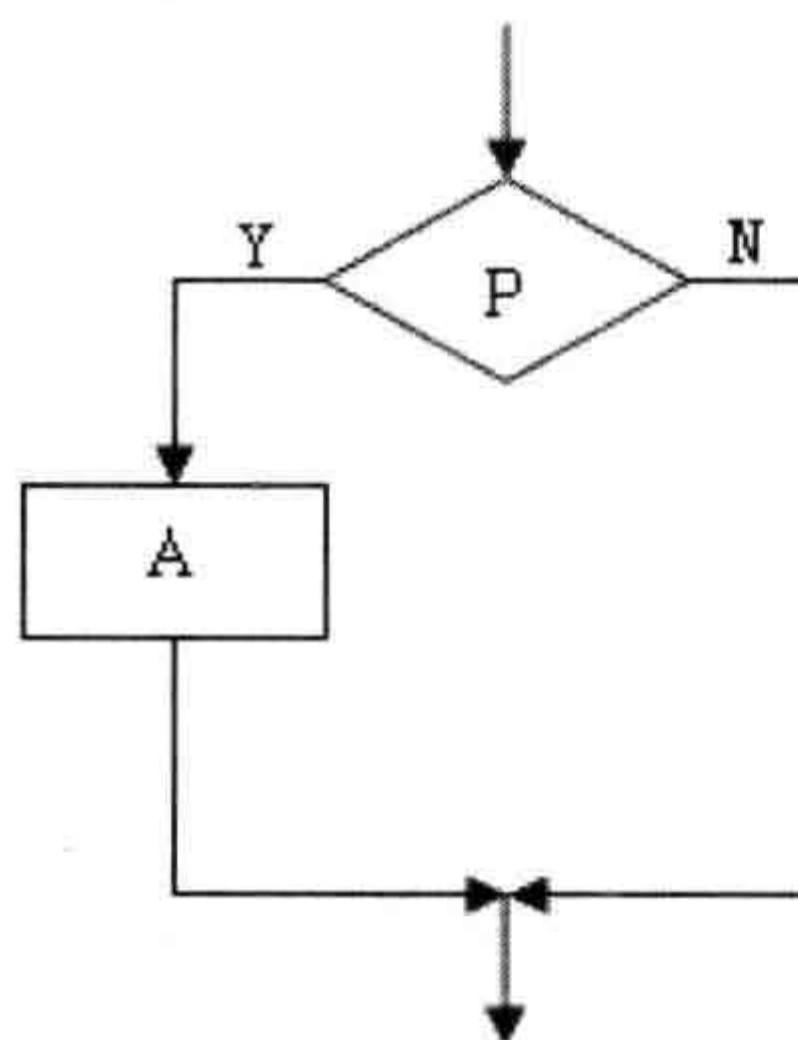


图 3.13 选择结构 2

图 3.12 表示的是根据给定的条件 P 是否成立选择执行 A 框或者是 B 框。而图 3.13 则表示根据给定的条件 P 进行判断,如果条件成立执行 A 框,否则什么也不做。

3. 循环结构

循环结构又称作重复结构,反复执行某一部分的操作,直到不满足条件时才终止循环。按照判断条件出现的位置划分,可将循环结构分为当型(`while`)循环和直到型(`until`)循环。

(1) 当型循环。当型循环是先判断条件 P 是否成立,如果成立,则执行 A 框,执行完 A 框后,再判断条件 P 是否成立,如果成立,接着再执行 A 框,如此反复,直到条件 P 不成立为止,此时不执行 A 框,跳出循环。当型循环如图 3.14 所示。

(2) 直到型循环。直到型循环是先执行 A 框,然后再判断条件 P 是否成立,如果条件 P 成立则再执行 A ,然后再判断条件 P 是否成立,如果成立,接着再执行 A 框,如此反复,直到条件 P 不成立,此时不执行 A 框,跳出循环。直到型循环如图 3.15 所示。





Note

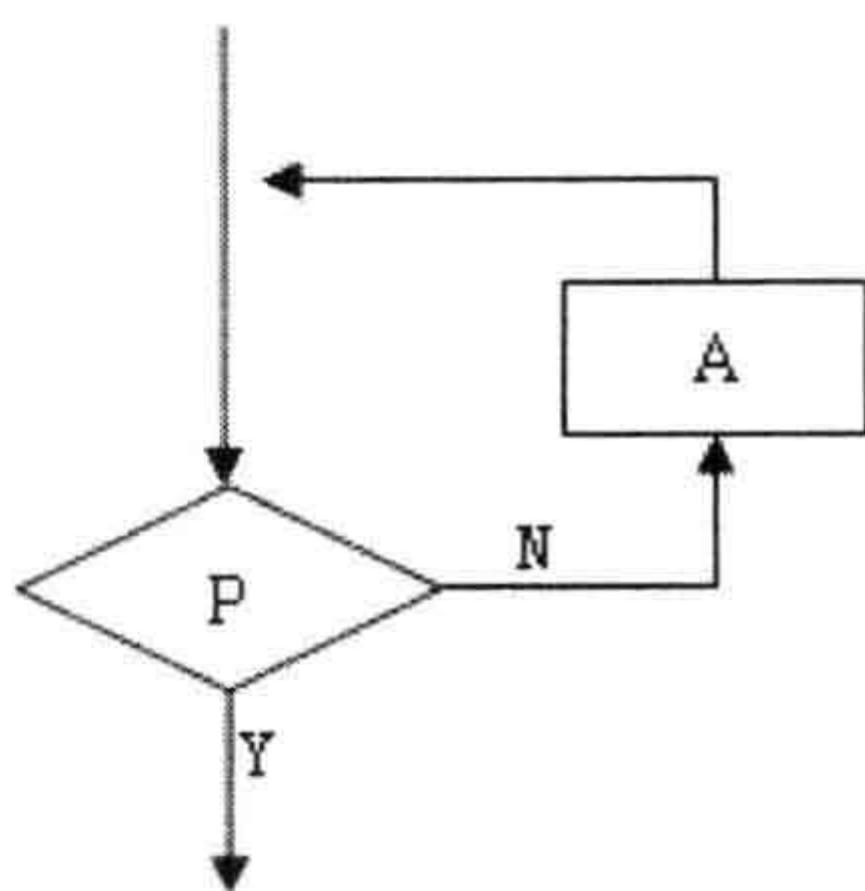


图 3.14 当型循环

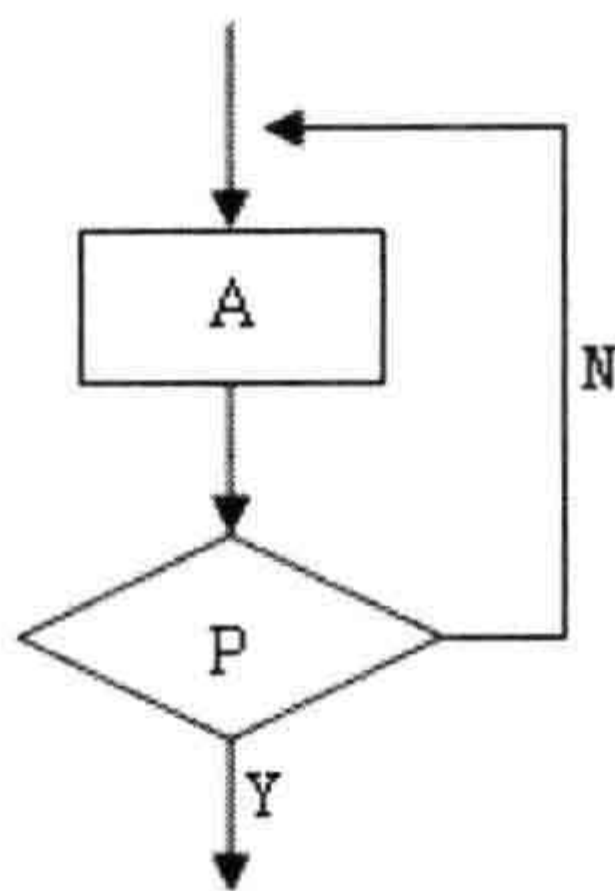


图 3.15 直到型循环

专家点评

这三种结构也是所有程序设计的三种基本结构。在算法设计中，掌握这三种设计结构是十分重要的。在后序章节中有关于程序的三种结构的详细介绍。

问题 26 算法有哪几类？

问题阐述

按照执行功能的不同，可以将算法分为不同的类别，那么算法有哪几类？

专家解答

计算机上的算法按照实现功能可以分为两大类：即数值型算法和非数值算法。数值算法主要解决一些工程中的数值计算问题，例如，数值积分、求解微分方程和定理猜想证明等。计算机专业课中，关于数值计算问题的《计算方法》等课程就是专门用来讨论数值算法的。非数值算法，主要用于解决那些非数值的计算机问题，例如，游戏算法和加解密算法等。

专家点评

这里是根据解决的问题进行分类的。在实际开发过程中，不必在意算法的分类问题，只要使用合适的算法解决问题即可。

问题 27 算法的效率度量方法有哪些？

问题阐述

衡量一个算法在计算机上的执行时间也称为算法的效率度量，那么算法的效率度量都有哪些方法呢？



专家解答

度量算法效率的方法有两种：

一种是事后计算的方法，即先实现算法，然后运行程序，测算其时间和空间的消耗。这种度量方法有很多弊端，由于算法的运行与计算机的软硬件等环境因素有关，不容易发现算法本身的优劣。同样的算法用不同的编译器编译出的目标代码不一样多，完成算法所需的时间也不同，并且当计算机的存储空间小时，算法运行时间就会延长。

一种是事前分析估算的方法，这种度量方法是通过比较算法的复杂性来评价算法的优劣。算法的复杂性与计算机软硬件无关，仅与计算时间和存储需求有关。算法复杂性的度量可以分为空间复杂度度量和时间复杂度度量。



Note

专家点评

一个算法采用不同的策略、不同的编译系统、不同的语言实现或者在不同的机器上运行，效率都有所不同。总的来说，算法的效率可以由问题的规模来衡量，设计算法应该尽量使用高效率低存储量需求的算法。

问题 28 什么是算法的时间复杂度？

问题阐述

算法的时间复杂度是评测算法好坏的主要指标，那么什么是算法的时间复杂度呢？

专家解答

算法的时间复杂度度量主要是计算一个算法所用的时间，算法所用的时间主要包括程序编译时间和运行时间。由于一个算法一旦编译成功就可以多次运行，因此忽略编译时间，在这里只讨论算法的运行时间。

算法的运行时间依赖于加减乘除等基本的运算、参加运算的数据的大小、计算机硬件以及操作环境等等。要想准确地计算时间是不可行的。而影响算法时间最为主要的因素是问题的规模，也就是输入量的多少。同等条件下，问题的规模越大，运行的时间也就更长。例如，求 $1+2+3+\cdots+n$ 的算法，即 n 个整数的累加求和，这个问题的规模为 n 。因此，运行算法所需的时间 T 是问题规模 n 的函数，记作 $T(n)$ 。

为了客观地反映一个算法的执行时间，通常用算法中基本语句的执行次数来度量算法的工作量。而这种度量时间复杂度的方法得出的不是时间量，而是一种增长趋势的度量。即当问题规模 n 增大时， $T(n)$ 也随之变大。换言之，当问题规模充分大时，算法中基本语句的执行次数为在渐进意义下的阶，成为算法的渐进时间复杂度，简称时间复杂度，通常用大 O 记号表示。用数学语言通常描述为：若当且仅当存在正整数 c 和 n_0 ，对于任意 $n \geq n_0$ ，都有 $T(n) \leq c \times f(n)$ ，则称该算法的渐进时间复杂度为 $T(n) = O(f(n))$ （或称算法在



$O(f(n))$ 中)。

对于某些算法即使问题规模相同,如果输入数据不同,则算法运行时间也不同。因此要想全面分析一个算法,需要考虑算法在最好、最坏、平均情况下的时间消耗。由于最好情况出现的概率太小,因此不具代表性,但是,当最好情况出现的概率大时,应该分析最好情况;虽然最坏情况出现的概率也太小,不具代表性,但是分析最坏情况能够让人们知道算法的运行时间最坏到什么程度,这一点在实时系统中很重要;分析平均情况是比较普遍的,特别是同一个算法要处理不同的输入时,通常假定输入的数据是等概率分布的。

通过对算法时间复杂度的分析,我们可以总结出这样一条结论:在计算任何算法的时间复杂度时,可以忽略所有低次幂和最高次幂的系数,这样可以简化算法分析,并使注意力集中在增长率上。

下面求一个程序段的时间复杂度,体会一下时间复杂度的计算方法。

例如,求下面程序段的时间复杂度。

```
for(i=1;i<7;i++)
    sum+=i;
```

解: $\text{sum}+=i$ 是基本语句,执行次数为 6,时间复杂度为 $O(1)$,此时间复杂度称为常量阶。

注意:

只要 $T(n)$ 不是问题规模 n 的函数,而是一个常数,它的时间复杂度均为 $O(1)$ 。

专家点评

在科技高度发展的今天,速度是一切事物的标准。高效的算法首先要求具有低的时间复杂度,但是也要综合考虑,不能为追求低的时间复杂度而忽略了其他标准。

问题 29 什么是算法的空间复杂度?

问题阐述

常常用算法的空间复杂度来评价算法的性能,那么什么是算法的空间复杂度呢?

专家解答

算法的空间复杂度是指在算法的执行过程中,需要的辅助空间数量。辅助空间数量指的不是程序指令、常数、指针等所需要的存储空间,也不是输入数据所占用的存储空间,而是除算法本身和输入输出数据所占据的空间外,算法临时开辟的存储空间。算法的空间复杂度分析方法同算法的时间复杂度相似,设 $S(n)$ 是算法的空间复杂度,通常可以表示为:

$$S(n)=O(f(n))$$



Note

**注意:**

在算法的时间复杂度和空间复杂度中,我们更注重算法的时间性能。因此在本书的算法分析中,不特别指明,均是对算法时间性能的分析。

专家点评

由于计算机硬件技术的发展,以及计算机磁盘存储空间不断扩大,算法的空间复杂度已经不那么重要了,但是也应该尽量减小算法的空间复杂度。



Note

问题 30 什么是分治法算法思想?

问题阐述

分治与递归就像一对孪生兄弟,在设计算法时经常是同时应用的,递归算法比较好理解,那么什么是分治法算法思想呢?

专家解答

分治法算法的设计思想就是将一个难以直接解决的大问题,分割成一些规模较小的相同问题,以便各个击破,分而治之。由分治法产生的子问题往往是原问题的缩小版,这样就为使用递归技术提供了方便。

分治法算法的基本思想就是将一个 n 规模的问题分解成 k 个规模较小的子问题,而且这些子问题都是独立的。除此之外,这些子问题除了规模比原问题小外,其他都是与原问题相同的。这样递归地解决这些子问题,然后将这些子问题的解合并,就可以得到原问题的解。

专家点评

分治法算法可以解决很多问题,在开发中经常被应用。例如,可解决棋盘覆盖问题、实现二分搜索技术等。

第 4 章

常用数据类型

- ▶▶ 声明变量和定义变量的区别是什么?
- ▶▶ 在开发时如何决定使用哪种数据类型?
- ▶▶ 什么是常量? 如何区分常量和变量?
- ▶▶ 各种数据类型所占的内存是多少?
- ▶▶ 字符与字符串的差别有哪些?
- ▶▶ 变量是否必须初始化?
- ▶▶ 为什么会发生数据溢出? 如何避免数据溢出?
- ▶▶ 局部变量和全局变量能否重名?
- ▶▶ 全局变量可不可以定义在可被多个.C 文件包含的头文件中? 为什么?
- ▶▶ 如何引用一个已经定义过的外部变量?
- ▶▶ 全局变量和局部变量的存储方式有什么区别?
- ▶▶ 整型常量的存储形式是怎样的?
- ▶▶ 整型常量的表示形式有哪几种?
- ▶▶ 使用了没定义的变量会有什么现象?
- ▶▶ static 关键字有什么作用?
- ▶▶ const 关键字有什么作用?
- ▶▶ const 与#define 相比有何优点?
- ▶▶ sizeof 不是函数吗?
- ▶▶ float 类型数如何与 0 值比较?
- ▶▶ 静态变量与自动变量的区别有哪些?



问题 31 声明变量和定义变量的区别是什么?

问题阐述

什么是声明变量, 什么是定义变量, 它们有什么区别?

专家解答

1. 什么是定义, 什么是声明

(1) 在 C 语言中, 使用变量之前必须先定义变量。所谓定义变量, 就是编译器创建了一个变量, 为这个变量分配一块内存并命名(变量名)。例如, 定义整型变量 a。

```
int a;
```

这条语句说明了 a 是一个整型变量, 编译器将为其分配一块大小为 int 型数据所占的内存空间。a 没有被赋初值, 它的初值默认为 0。在定义的同时, 也可以为其赋值进行初始化。如:

```
int a=1;
```

这条语句不仅说明了编译器为 a 分配了内存, 还说明了在整型变量 a 分配的内存中存储的值。

注意:

不应该在头文件中定义变量, 因为一个头文件可能会被一个程序的许多源文件所包含。

(2) 所谓声明, 就是告诉编译器变量的类型, 编译器并不为其分配内存, 此变量已经定义过, 故声明可以多次进行。例如, 声明外部变量 a。

```
extern int a;
```

这条语句说明了 a 是一个外部整型变量, 并且它的存储空间是在程序的其他区域分配的。extern 置于变量前, 以标示变量的定义在别的区域中, 下面的代码用到的变量 a 是外部的, 不是本区域定义的, 提示编译器遇到变量 a 在其他区域中寻找其定义。

2. 声明变量和定义变量的区别

(1) 定义创建了变量, 并为其分配内存; 声明没有分配内存。

(2) 一个变量在一定的区域内只能被定义一次, 却可以被多次声明。

专家点评

通过上面的学习, 可以知道什么是声明、什么是定义, 它们看似差不多, 其实还是有一定的差别的。一个分配内存(定义), 一个不分配内存(声明)。

学习 C 语言, 首先就要弄懂定义和声明的含义, 这样可以避免重复定义覆盖原始值,



Note



从而提高程序的运行效率。

问题 32 在开发时如何决定使用哪种数据类型?

问题阐述

程序中的每个数据都必须有明确的数据类型,那么在开发时如何决定使用哪种数据类型呢?

专家解答

如果需要的数大于 32767 或者小于-32767,就应该使用 long 型。

如果有大数组或很多结构,就使用 short 型。除了上述两种情况外,使用 int 型。

如果严格定义的溢出特征很重要而负值无关紧要,或者期望在操作二进制位和字节时避免符号扩展的问题,就使用对应的无符号类型。需要注意的是,在表达式中混用有符号和无符号值的情况。

对于字符型可以当作小的整型使用,但是由于不可预知的符号扩展和代码的增加,会使数据丢失,此时使用无符号字符型会更恰当些。

如果需要使用小数,就使用浮点类型。如果要求小数点后精确至少 10 位,就使用 double 型;如果要求小数点后更加精确,即有效位超过 10 位,那么就使用 long double 型。除了这两种情况外,使用 float 型。

注意:

unsigned 和 signed 不能修饰浮点类型。

专家点评

从上面的讲解中可以知道什么情况使用什么类型的数据。在 C 语言程序中,对于特定的数值应定义相应的数据类型,因此要了解各个数据类型的范围,才有利于数据的存储与运算。

问题 33 什么是常量? 如何区分常量和变量?

问题阐述

什么是常量,什么是变量? 怎样区分二者?

专家解答

1. 常量与变量

常量即其值在程序运行的过程中是不可以改变的,如 123, -4567 为数值常量;变量是在程序运行期间其值是可以进行变化的量,如 int a; char b; 是整型变量 a 与字符型变量 b。



2. 常量与变量的区别

常量不能被修改，其存放的位置未知；而变量随时都可以进行修改，且有特定的存储空间。

常量与变量的最大区别就是变量有变量名，而常量只有符号没有名字。

说明：

定义常量要使用预处理命令，在系统编译之前由编译系统来处理，将不可以在程序运行中更改的数据定义为常量，这样不但增强程序的可读性，还增强了程序的可修改性。而定义变量，要给出变量的数据类型和变量名，并且要符合标识符的命名规则。

**Note**

专家点评

根据上面的解答，我们知道了变量和常量的特点，根据这些特点便可以对二者进行区分。

问题 34 各种数据类型所占的内存是多少？

问题阐述

在 C 语言中，long、int 和 short 这三种类型数据所占用的内存各是多少？

专家解答

这 3 种类型的数据所占用的内存是由所用机器的机器字长决定的。在 16 位机中，long 型数据所占用的字节是 4，int 型数据所占用的字节是 2，short 型数据所占用的字节是 2；而在 32 位机上，long 型数据所占用的字节是 4，int 型数据所占用的字节是 4，short 型数据所占用的字节是 2。

说明：

无论在什么环境下，都有：short 型数据所占用内存不得长于 int 型数据，int 型数据所占用内存不得长于 long 型数据。

专家点评

从上面的描述可以知道，不同的类型占用的内存空间不同，保存数据的范围也就不同，可以根据自己的需要选择不同的数据类型。而在不同的环境中，同一种类型的内存不同，所以在选择数据类型时，一定要了解自己使用的运行环境，才可以避免数据的溢出或者造成空间的浪费，从而提高程序的运行效率。



问题 35 字符与字符串的差别有哪些？



问题阐述

‘a’ 与 “a” 有什么不同？

专家解答

‘a’ 为字符常量，“a” 为字符串常量。字符常量与字符串常量的差别主要有以下 3 点：

(1) 定界符的使用不同。字符常量使用的是单引号，而字符串常量使用的是双引号。‘a’ 是字符常量，而 “a” 是字符串常量。

可以用字符型常量来赋值，例如：

```
char b;  
b='a';
```

上述赋值是正确的，但是下面的赋值都是错误的。

```
char b;  
b="a";
```

或者

```
b="Hello";
```

注意：

编写 C 语言程序的时候，千万不要把一个字符串常量赋值给一个字符变量，否则会造成字符丢失。

(2) 长度不同。字符常量只能有一个字符，也就是说，字符常量的长度为 1。而字符串常量的长度却可以是 0，即使字符串常量中的字符数量也只有 1 个，但是长度却不是 1。字符串常量 “a”，其长度为 2。为什么字符串常量 “a” 的长度为 2？这是因为系统会自动在字符串常量尾部加上一个转义字符 ‘\0’，作为结束标志，故 “a” 的长度是 2，而 ‘a’ 的长度是 1。

(3) 存储的方式不同。在字符常量中存储的是字符的 ASCII 码值，而在字符串常量中，不仅要存储有效的字符，还要存储结尾处的结束标志 ‘\0’。

专家点评

通过上面的学习可以知道字符与字符串的本质区别：字符常量是单个字符，而字符串可以是单个字符，也可以是多个字符。需要注意的是，不要将单引号和双引号弄混了，因



为用单引号括起来的一个字符代表一个整数，而用双引号括起来的一个字符代表一个指针。若将两者混用，将产生难以预料的错误。

问题 36 变量是否必须初始化？



Note

问题阐述

在使用变量时，首先要对其定义，然后进行初始化，那么变量是否必须初始化呢？

专家解答

1. 问题分析与解答

在 C 语言程序中，通常需要对一些变量设定初值。C 语言允许在定义变量的同时给变量赋初值，这个过程就是变量的初始化过程。

例如：

```
int a=1;
float b=3.14;
char c='v';
```

上面的代码分别对 a, b, c 进行初始化。还可以将其写成：

```
int a;
a=1;
float b;
b=2;
char c;
c='v';
```

这段代码分别对 a, b, c 进行赋值，而不是进行初始化。

看下面的一段代码。

```
int t,a=2,b=3;
t=a;
a=b;
b=t;
```

首先对 a 和 b 进行初始化，然后对 t 进行赋值操作，这里 t 不用初始化。

注意：

若对几个变量赋相同的初值，应写成：“int a=1,b=1,c=1;”而不能写成“int a=b=c=1;”。

2. 问题的扩展

赋值与赋初值有什么不同？



二者的区别在于：为变量赋值是通过赋值表达式在运行期间动态赋值，而为变量赋初值则是在定义变量的同时在编译时静态赋值。如对 a 进行赋值，对 b 进行赋初值，形式如下。

```
int a,b=3;
a=2;
```

为变量赋值占用的是运行时间，而为变量赋初值占用的是编译时间。

专家点评

从上面的分析得知，变量不是一定要初始化的，也可以先进行定义，再进行赋值，这和初始化的效果是一样的。但是如果提高运行效率，就得对变量进行初始化。

问题 37 为什么会发生数据溢出？如何避免数据溢出？

问题阐述

有以下程序：

```
#include <stdio.h>
void main()
{
    int i,rst;                /*声明变量*/
    i=32767;                 /*定义变量*/
    rst=i+1;                 /*变量值加一*/
    printf("%d,%d",i,rst);   /*输出结果*/
}
```

运行后 rst 的结果是什么？

专家解答

1. 问题分析

按正常的计算，rst 应该是 32768，但是运行后 rst 的结果是-32768。这是什么原因呢？

在 C 语言中可以使用各种类型的标识符，但是 C 语言标准里并没有规定这些类型的具体长度，要由各个 C 编译系统自己规定。一般以一个机器字存放一个 int 数据，早期的计算机字长一般为 16 位，故以 16 位存放一个整数，整数的范围在-32768~32767 之间，整数范围太小，在使用的时候就容易产生溢出现象。但现代的计算机一般都为 32 位以上，以 32 位存放一个整数，范围可达 $\pm 2.1 \times 10^9$ ，所以当前一般将 int 和 long 都定义为 32 位。在写程序时，要了解所用系统对标识符长度的规定，以免出现上面的错误。本程序使用 Toubo C 进行编译，在 Toubo C 中，一个 int 型变量的最大允许值为 32767，如果再加 1 则“溢出”。

如果真的产生“溢出”，运行时也不会报错，而是将结果从“头”开始计算，即是上面的运行结果-32768，这与编程者的原意不同。由于系统不会给出“出错信息”，所以要靠程序员的细心和经验来进行排错。



说明:

当前大多编译器多将 int 规定为 32 位, 所以一般不会出现“溢出”现象。

2. 问题的解决方法

将数据定义为长整型 long 类型, 在输出的时候使用“%ld”即可。

注意:

long 型数据可以得到大范围的整数, 但同时会降低运算速度。因此, 除非不得已, 否则不要随便使用 long 型。



Note

专家点评

C 语言数值数据类型都有一个取值范围。在进行数值运算的时候, 有时会因数值超出了定义类型的取值范围而产生溢出的情况。所以在表示数据时, 应根据需要选择适当的数据类型。

问题 38 局部变量和全局变量能否重名?

问题阐述

局部变量和全局变量能否重名?

专家解答

全局变量和局部变量是按照变量的作用域划分的。简单地说, 局部变量是定义在函数内部的变量; 全局变量是定义在函数之外的变量。全局变量可以为本文件中其他函数所共用。

局部变量和全局变量可以重名, 局部变量会屏蔽全局变量。要使用全局变量, 要在变量名前添加“::”。

专家点评

从上面的解答可以得知, 局部变量和全局变量是可以重名的, 因为它们的作用域不同。在函数中, 默认使用的是局部变量。

问题 39 全局变量可不可以定义在可被多个.C 文件包含的头文件中? 为什么?

问题阐述

全局变量可不可以定义在可被多个.C 文件包含的头文件中? 为什么?



专家解答

全局变量可以定义在可被多个.C 文件包含的头文件中,在不同的.C 文件中以 `static` 形式声明同名全局变量,前提是其中只能有一个.C 文件中对此变量赋初值,此时连接才不会出现错误。



Note

专家点评

从上面的解答中知道,可以在被多个.C 文件包含的头文件中定义全局变量,只是不可在多个.C 文件中对全局变量赋初值,否则会造成连接错误。

问题 40 如何引用一个已经定义过的外部变量?

问题阐述

如何引用一个已经定义过的外部变量?

专家解答

如果在一个文件中定义了一个外部变量 `a`,在另一个程序文件中再定义一个外部变量 `a`,就会产生一个“重复定义”的错误,那么怎样引用已经定义的外部变量呢?

引用被定义过的外部变量有两种方式。一是用引用头文件的方式,二是用 `extern` 关键字来引用已经定义过的外部变量。

说明:

使用引用头文件的方式来引用某个头文件中的外部变量,如果变量名书写错误,则在编译时提示错误;使用 `extern` 关键字引用时,如果变量名书写错误,则在连接时提示错误。

使用 `extern` 关键字来引用已定义过的外部变量 `a` 的方法是:首先在一个文件中声明这个外部变量,然后在另一个文件中使用 `extern` 关键字对这个变量 `a` 进行外部变量声明。这样在程序编译连接的时候,编译器就会知道变量 `a` 是一个已经定义过的外部变量,并且将外部变量 `a` 的作用域扩大到该文件,在该文件中可以合法地使用这个外部变量 `a`。

专家点评

从上面的解答中可知,引用已经定义过的外部变量可以用 `extern` 关键字,但是需要注意的是,一定要慎重使用多文件的外部变量,因为多文件的外部变量是在多个文件中被使用的,在某个文件的函数中被调用一次,变量的值就改变一次,这样就会影响到下一次对该变量的调用。



问题 41 全局变量和局部变量的存储方式有什么区别?

问题阐述

全局变量和局部变量在内存中是怎样存放的? 两者之间有何区别?

专家解答

全局变量储存在静态数据库中, 局部变量存储在堆栈中。全局变量在程序开始执行时分配存储区, 程序执行完毕释放, 在程序执行过程中全局变量始终占据固定的存储单元; 局部变量是动态分配存储空间的, 在调用变量所在函数时, 系统会给函数的局部变量分配存储空间, 在函数调用结束时自动释放这些存储空间。

专家点评

本题需要掌握的是局部变量与全局变量的存储方式以及生命周期, 这些都是需要记住的基本内容。只有掌握了这些知识, 才能真正掌握局部变量与全局变量的使用。

问题 42 整型常量的存储形式是怎样的?

问题阐述

-8 在内存中的存储形式是怎样的?

专家解答

整型数据在内存中以二进制的形式存放, 数值是以补码表示的。一个正数的补码和其原码的形式相同, 一个负数的补码是将该数绝对值的二进制形式, 按位取反再加 1。这里 -8 的绝对值在内存中的存储形式如图 4.1 所示。

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

图 4.1 十进制 8 在内存中的存储

将其进行取反操作, 得到的结果如图 4.2 所示。

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

图 4.2 按位取反的结果

取反后加 1, 结果如图 4.3 所示。





1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

图 4.3 取反加一（-8 的存储效果）

专家点评

整型常量在内存中以二进制形式存储的有两种数据：有符号和无符号。上面介绍的是有符号的数据在内存中的存储形式，无符号的整型常量是正数或者 0，所以最高位不用来存放数据的符号，而是用来存放数据本身。

问题 43 整型常量的表示形式有哪几种？

问题阐述

数值 029 是一个什么样的数？

专家解答

整型常量有三种不同的表达方式：八进制、十六进制和十进制。

八进制整常数是以前缀 0 作为前缀，数码取值为 0~7，如 0452 和 0761。十六进制整常数是以前缀 0X 或 0x 作为前缀，其数码取值为 0~9、A~F 或 a~f，如 0x145 和 0x23fc。十进制整常数无前缀，其数码为 0~9，如 123 和 908。

数值 029 是以 0 开头的，符合八进制形式的第一个条件，但含有数字 9，超出了数码取值的范围，所以它不是八进制数，因此它是非法的数。

专家点评

如果对整型常量的三种表达形式熟悉，就不难判断出 029 是个非法的数。掌握了整型常量的表达形式就可以对整型常量进行存储以及运算。

问题 44 使用了没定义的变量会有什么现象？

问题阐述

变量在使用前都要进行定义，若没进行定义就使用，会出现什么现象？

专家解答

1. 问题分析

首先看一个例子，代码如下。



Note

```
#include <stdio.h>
void main()
{
    int x = 10;                /*定义变量*/
    while( x > 0 )             /*循环*/
    {
        x--;                  /*自减*/
        sum=sum+x;            /*加法运算*/
    }
    printf("%d\n",sum);        /*输出*/
}
```

运行程序执行上面的代码，会出现一个编译错误，如图 4.4 所示。

error C2065: 'sum' : undeclared identifier

图 4.4 编译错误

变量是用于存储数据的。每个变量都有一个名字，在内存中占据一定的存储单元，在该存储单元中存放变量的值。在 C 语言中，要求对使用的变量做定义，就是要先定义，后使用。这样能够方便确定变量类型并为其分配存储单元，同时也便于在编译时检查该变量进行的运算是否合法。使用未被定义的变量，在编译时就会提示错误信息。

2. 问题的解决办法

将未定义的变量进行定义。代码如下。

```
int sum = 0;
int x = 10;
```

再次运行程序，结果如图 4.5 所示。

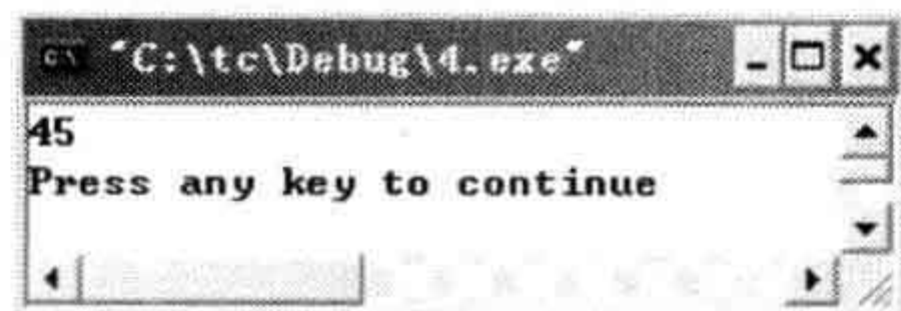


图 4.5 程序运行结果

专家点评

在使用变量之前必须定义变量，如果没有定义变量就使用，在程序编译时会提示错误信息，程序也不能被执行，这种错误是很容易被发现的。

问题 45 static 关键字有什么作用？

问题阐述

static 关键字经常被使用，那么它的含义是什么？有哪些作用呢？



专家解答



Note

static 关键字的主要作用如下。

(1) 函数体内, static 变量的作用范围为该函数体, 不同于 auto 变量, 该变量的内存只被分配一次, 因此其值在下次调用时仍维持上次的值。

(2) 在模块内的 static 全局变量可以被模块内所有函数访问, 但不能被模块外其他函数访问。

(3) 在模块内的 static 函数只可被这一模块内的其他函数调用, 这个函数的使用范围被限制在声明它的模块内。

下面举例进行分析。分析下面的代码, 看一下 static 关键字在不同位置定义的变量的值有什么变化。程序代码如下。

```
static int i;
void fun1(void)
{
    i = 0;
    i++;
    printf("i=%d ",i);
}
void fun2(void)
{
    static int j = 0;
    j++;
    printf("j=%d\n",j);
}
int main()
{
    int n;
    for(n=0; n<10; n++)
    {
        fun1();
        fun2();
    }
    return 0;
}
```

上面的代码定义了一个全局型的静态变量 i 和一个局部静态变量 j。每次调用函数的时候, 改变变量的值, 使其累加。由于 i 在每次调用函数的时候都赋予了新值, 所以每次的值都是 1; 因为 j 被定义为局部静态变量, 其值在每次函数结束时都被保留, 所以结果都是上一次值加 1。运行结果如图 4.6 所示。

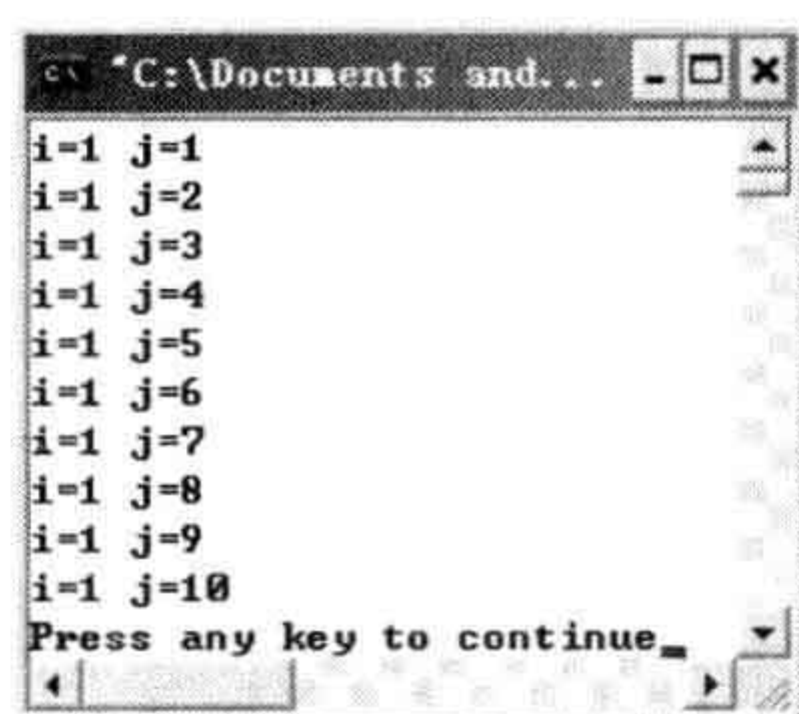


图 4.6 static 关键字的应用



Note

专家点评

静态全局变量只能在变量被定义的文件中使用，其他文件即使用 `extern` 声明也没法使用它。准确地说，作用域是从定义之处开始，到文件结尾处结束，在定义之处前面的那些代码行也不能使用它，想要使用就得在前面再加 `extern`。

问题 46 const 关键字有什么作用？

问题阐述

const 关键字有什么作用？

专家解答

(1) 欲阻止一个变量被改变，可以使用 `const` 关键字。在定义该 `const` 变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了。

(2) 对指针来说，可以指定指针本身为 `const`，也可以指定指针所指的数据为 `const`，或二者同时指定为 `const`。

(3) 在一个函数声明中，`const` 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值。

下面举例进行分析。分析下面的代码，找出程序中的错误。程序代码如下。

<code>#include<stdio.h></code>	第 1 行
<code>void main()</code>	第 2 行
<code>{</code>	
<code>const int a=5;</code>	第 3 行
<code>int b=6;</code>	第 4 行
<code>int c=7;</code>	第 5 行
<code>const int* m=&b;</code>	第 6 行
<code>int *const n=&b;</code>	第 7 行
<code>const int* const k=&b;</code>	第 8 行
<code>a=2;</code>	第 9 行
<code>m=&c;</code>	第 10 行
<code>*m=1;</code>	第 11 行



Note

n=&c;	第 12 行
*n=1;	第 13 行
k=&c;	第 14 行
*k=1;	第 15 行
}	

第 9 行, 由于变量 `a` 为 `const` 修饰的, 所以它不可改变, 这样会出现编译错误。如在第 3 行没有对 `a` 进行初始化, 系统将为 `a` 赋一个随机数, 在后面也是不可以进行赋值的。

第 10 行和第 11 行, `m` 定义为 `const int*` 类型, `const` 在 `int*` 的左侧, 它是用修饰指针所指向的变量, 也就是说指针指向为常量, `m` 指向变量 `c` 是允许的, 因为这样并没有修改指针 `m` 所指的内容, 而是修改指针本身 `m`。但是第 12 行, 修改的却是指针 `m` 所指向的内容, 这将被不允许, 编译出错。

第 12 行和第 13 行, `n` 定义为 `int* const` 类型, `const` 在 `int*` 的右侧, 它修饰指针本身, 即修饰常量。因此, 第 12 行中修改指针 `n` 是不被允许的, 而第 12 行修改的是指针指向的内容, 是被允许的。

第 14 行和第 15 行, `k` 定义为 `const int* const` 类型, `int*` 的左右两侧都有一个 `const`, 它表示既不能修改指针, 也不能修改指针指向的内容, 故这两行都会出现编译错误。

注意:

变量 `a`、变量 `n` 和变量 `k` 在声明的同时一定要进行初始化, 因为它们在后面都不能被赋值, 变量 `m` 可以在声明时不进行初始化。

专家点评

(1) 精确地说, `const` 修饰的是只读变量, 只读变量的值在定义后就不能再改变了。

(2) 在定义 `const` 变量时, 必须要对该变量初始化, 否则编译器会报错。`const` 变量还可以避免只读变量被意外修改。在程序中, 如果修改了 `const` 定义的只读变量, 则编译器就会报错。

(3) `const` 限定的对象通常是在运行时不能被赋值的对象, 因此用 `const` 限定的对象的值并不是一个真正的常量, 不能用作数组维度等。

问题 47 `const` 与 `#define` 相比有何优点?

问题阐述

`const` 和 `#define` 修饰的都是不可变的量, 请说出二者相比有何优点?

专家解答

(1) `const` 修饰的只读变量具有特定的数据类型, 而宏没有数据类型。编译器可以对



前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

(2) 有些集成化的调试工具可以对 `const` 常量进行调试，但是不能对宏常量进行调试。

(3) 编译器通常不为普通 `const` 只读变量分配存储空间，而是将它们保存在符号表中，这使它成为一个编译期间的值，没有了存储与读内存的操作，使得它的效率也更高。

例如：

```
#define P 3.14          /*宏常量*/
const int G=5;          /*此时并未将 G 放入内存中*/
int i=G;                /*此时 G 分配内存，以后不再分配*/
int I=P;                /*预编译期间进行宏替换，分配内存*/
int j=G;                /*没有内存分配*/
int J= P;               /*再进行宏替换，又一次分配内存*/
```

`const` 定义的只读变量从汇编的角度来看，只是给出了对应的内存地址，而 `#define` 给出的是立即数，所以，`const` 定义的只读变量在程序运行过程中只有一个备份（因为它是全局的只读变量，存放在静态区），而 `#define` 定义的宏常量在内存中有若干个备份。

`#define` 宏是在预编译阶段进行替换，而 `const` 修饰的只读变量是在编译的时候确定其值。

`#define` 宏没有类型，而 `const` 修饰的只读变量具有特定的类型。

专家点评

本题主要从节省空间、提高效率、避免错误等方面解答了 `const` 和 `#define` 比较的优点。

说明：

`const` 修饰的不是常量，而是只读变量，并且这个只读变量不能用作数组的维度，也不能放在 `case` 关键字后。

问题 48 sizeof 不是函数吗？

问题阐述

`sizeof` 怎么用，它不是函数吗？

专家解答

1. `sizeof` 的使用

首先，看一个例子，说出下列代码中各个变量在内存中的大小（在 VC++6.0 环境下计算）。



Note



Note

```
char* s1="123456789";
char s2[]="123456789";
char s3[100]="123456789";
int s4[100];
char c1[]="abc";
char c2[]="a\n";
char* c3="a\n";
```

分析与解答:

s1 是一个字符指针, 指针的大小是一个确定的值, 就是 4, sizeof(s1)=4。

s2 是一个字符数组, 这个数组最初的大小未定, 填充值是“123456789”, 一个字符占一位, 再加上隐含的“\0”一共是 10 位, 即 sizeof(s2)=10。

s3 是一个字符数组, 这个数组开始预分配 100, 所以它的大小是 100 位, 即 sizeof(s3)=100。

s4 是一个整型数组, 但是每个整型变量所占空间是 4, 所以它的大小一共是 400 位, 即 sizeof(s4)=400。

c1 与 s2 类似, 占 4 位。c2 里面有一个“\n”, “\n”为转义字符, 占一位, 加上隐含的“\0”, 大小一共是 3 位。c3 是一个字符指针, 指针的大小是定值 4。

2. sizeof 不是函数吗

通过下面这个例子来说明 sizeof 是否为函数。

```
int i=0;
```

下列哪个不能运行出结果?

- A. sizeof(int);
- B. sizeof(i);
- C. sizeof int;
- D. sizeof i;

分析与解答:

sizeof 表示的是计算对象所占内存空间的大小。“sizeof()”和函数的形式一样, 函数名后面跟了一对括号, 但实际上它只是关键字, 并非函数。本例子中, 选项 D 的运行结果证明它不是函数, 去掉括号也能运行出结果, 但是函数却不能将括号去掉。

另外, 需要注意的是, sizeof 在计算变量所占空间大小时, 括号是可以省略的, 而在计算类型大小时, 括号则不能省略。故 A、B、D 都能运行出结果“4”, 而只有 C 不能运行出结果。

专家点评

从上面的例子可以知道, sizeof 不是函数, 它只是关键字, 但是一般情况下, 它将以“sizeof()”貌似函数的形式出现。



问题 49 float 类型数如何与 0 值比较?

问题阐述

写出 float a 与“0 值”比较的 if 语句。

专家解答

1. 问题分析与解答

一般地, 如果用 if 判断一个数值型变量(short、int、long 等), 应该用 if(a==0), 表示的含义是 a 与 0 进行“数值”上的比较; 但 float 型变量并不精确, 不能直接拿来与 0 进行比较, 所以不可使用“==”或“!=”这种形式, 应该使用“>=”或“<=”这种形式。如果写成 if(a==0.0), 显然不对。因此正确的写法为: if((a >= -0.00001) && (a <= 0.00001))。

2. 问题扩展

如果将两个浮点型的数值进行比较或计算, 会得到什么样的结果呢?

下面通过一个例子来说明这个问题, 代码如下。

```
#include<stdio.h>
void main()
{
    float a,b;
    a=3.1234567452e10;
    b=a+0.01;
    printf("a=%f\nb=%f\n", a, b);
}
```

程序运行结果如图 4.7 所示。

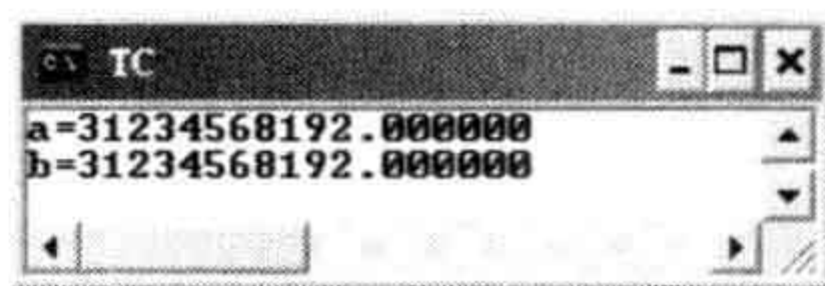


图 4.7 两个浮点数的计算

从运行结果可以看出, a 和 b 的值是相同的, b 的值并未增加。由于浮点型变量在内存中占用的存储空间是有限的, 故能提供有效数字是有限的, 计算时会舍去有效位后的数字全部舍去, 这样就存在了舍入误差, 程序中 a 的值比 b 大得多, a+0.01 的值应该是 31234567452.01, 但是 float 类型的有效数字是 7 位, 故出现 a 和 b 的值都是 31234568192 的结果。

注意:

在处理浮点型数据时, 尽量避免将很大的数和一个很小的数进行计算, 否则会丢较小的那个数。



Note



专家点评



Note

通过上面的讲解可以知道，浮点型变量并不像整型变量那样精确，大多数情况下，浮点型只能是一种近似表示。本题是正是因为 float 和 double 类型的数据都是有精度限制的，所以直接拿来与 0.0 比较不会得到我们预期的较精确的数值。

问题 50 静态变量与自动变量的区别有哪些？

问题阐述

静态变量与自动变量的区别有哪些？

专家解答

(1) 静态存储类型的局部变量是在静态存储区内分配内存单元，在程序的整个运行期间内都不释放空间。而自动类型的局部变量属于动态存储类型，是在动态存储区内分配存储单元的，函数调用结束后存储单元即被释放。

(2) 静态局部变量是在编译时赋初始值，并且只赋一次初值，在以后每次调用函数时，都不再重新为其赋值，只是使用上一次函数被调用结束时变量的值。而自动局部变量的初值不是在编译时赋予的，而是在函数调用时赋予的，每调用一次，函数都对变量重新赋一次初值。

(3) 如果定义的静态局部变量没有对其进行赋值，则该变量的默认值为 0 或者为空字符串。而对于自动局部变量来说，如果不赋值，则变量的值是一个不确定的值。这是因为在函数被调用时，会为该变量分配一个存储空间，在函数结束时，存储空间被释放。在下次调用该函数的时候，又会重新分配一个存储空间，这两次分配的存储空间是不一样的，存储空间中的值也是不确定的。

例如，下面的代码。

```
void test()
{
    int a=1
    a=a+1
}
```

这段代码中，变量 a 的值是 1，即使多次调用以后，a 的值还是 1。再来看下面的代码。

```
void test()
{
    static a=1
    a=a+1
}
```




上述代码中, 变量 `a` 的值是随着调用次数的不同而不同的。如果该函数被调用了 3 次, 变量 `a` 的值就变为 4, 并且保持不变, 直到下次再调用这个函数。

专家点评

从上面的学习可以了解到, 虽然静态局部变量的值在函数调用结束以后也是保持不变的, 但是它不能被其他的函数所引用, 只能在所在的函数中使用。



Note

第 5 章

运算符与表达式

- ▶▶ 运算符的优先级和结合性是怎样的?
- ▶▶ 如何区分“,”是运算符还是分隔符?
- ▶▶ C语言如何解释 $x=a+=b+c$?
- ▶▶ $x=x+1, x+=1, x++$, 哪个效率最高?
- ▶▶ 什么是运算符的目? 怎样进行区分?
- ▶▶ 使用“++”和“--”运算符需要注意些什么?
- ▶▶ 如何理解 $i++j$?
- ▶▶ 赋值表达式中什么是左值和右值? 数组名作为左右值时又具有怎样的意义?
- ▶▶ 如何确定条件表达式的结果的数据类型?
- ▶▶ “%”运算符是否可以对小数进行运算?
- ▶▶ “/”运算符得到的结果一定为整数吗?
- ▶▶ 在进行多种数据类型混合运算的时候, 数据类型自动转换有哪些规则?
- ▶▶ C语言中有哪些简化的运算表达式?
- ▶▶ 使用逻辑表达式需要注意哪几点问题?
- ▶▶ $l++*i++$ 这样的表达式为什么不能得到预期的结果?
- ▶▶ $a[i]=i++$; 这样的代码正确吗?
- ▶▶ 编写表达式时需要注意什么?
- ▶▶ 如何理解 $c=a,b;?$
- ▶▶ 为无符号类型变量赋值时, 数据类型应怎样转换?
- ▶▶ C语言表达式的求值顺序总是按照运算符的结合性保证“自左至右”或者“自右至左”吗?



问题 51 运算符的优先级和结合性是怎样的？

问题阐述

各种运算符的优先级是怎样的？结合顺序又是怎样的？

专家解答

C 语言中的运算符种类丰富，有 40 多种，分为 15 种优先级，而且还具有结合性的特点。在表达式中，各运算对象参与运算的先后顺序不仅要遵守运算符优先级的规定，还有运算符结合性的制约，以便确定运算对象的执行顺序。这种结合性其他语言没有的。各种运算符的优先级从高到低的顺序如图 5.1 所示。

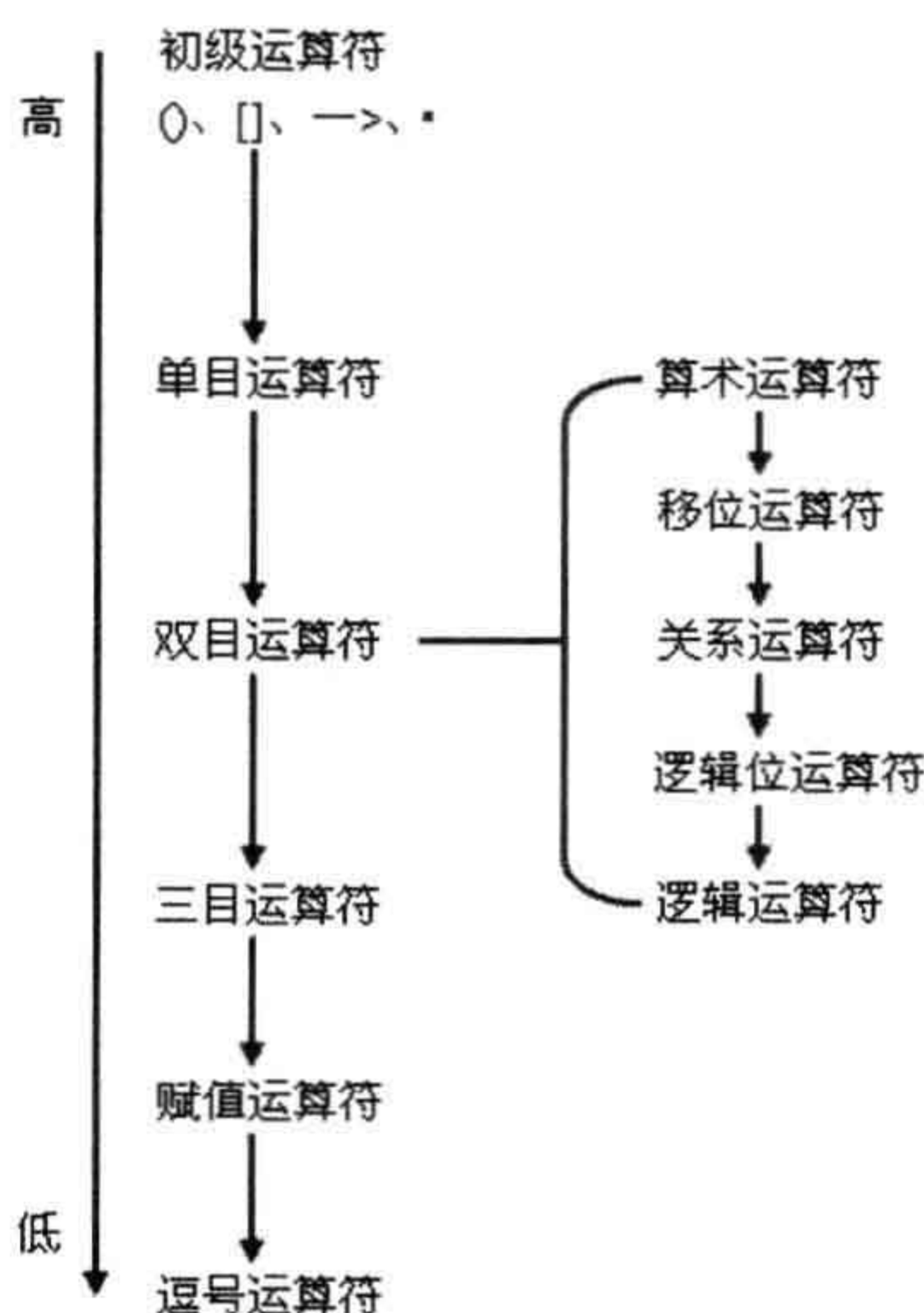


图 5.1 C 语言运算符的优先级

以上运算符的优先级由上到下递减。初级运算符优先级最高，逗号运算符优先级最低。同一优先级的运算符，运算次序由结合方向而定。

运算符的结合性：

C 语言中，各运算符的结合性分为两种，即自左至右结合和自右至左结合。例如，算术运算符的结合性是自左至右的。大家可以记住一个规律，单目运算符、条件运算符和赋值运算符是自右至左结合的，其他算术运算符都是自左至右结合的。

专家点评

每种运算符之间也有先后顺序。例如，算术运算符中就是“先乘除，后加减”，还可以使用圆括号对表达式进行结合的区分。



Note



问题 52 如何区分“,”是运算符还是分隔符?



问题阐述

Note

逗号在 C 语言中有时可以作为运算符来连接表达式,有时还可以作为分隔符,起到分隔的作用。那么,该如何区分逗号是运算符还是分隔符呢?

专家解答

“,”作为分隔符主要用于以下情况:

(1) 变量声明时,使用逗号分隔多个变量名。

(2) 函数有多个参数时,用逗号分隔参数。

逗号运算符是 C 语言提供的一种特殊的运算符,用它可以将两个表达式连接起来。例如,“1+2, 4+5”,称为逗号表达式,也称为“顺序求值运算符”。它的表达式一般形式为“表达式 1, 表达式 2”,执行顺序是先计算表达式 1 的值,再计算表达式 2 的值。表达式也可以扩展为“表达式 1, 表达式 2, 表达式 3, ..., 表达式 n”。

例如:

```
int min(a,b,c)
int a,b,c;
```

在这两句代码中,逗号都是起到分隔符的作用。在第一句代码中,逗号用来分隔多个函数参数。第二句代码中,逗号分隔多个变量名。

又如:

```
int x,y,z;
x=1, y=2, z=3;
```

这两句代码中,第一句中逗号起到的是分隔符的作用,第二句代码中的逗号是起到运算符的作用。

第二句代码页相当于下面的代码。

```
x=1; y=2; z=3;
```

一般在只允许出现一个表达式的地方却出现了多个表达式,并且是用逗号相连接的,这种表达式就是逗号表达式。此时,逗号就是运算符。

看下面这个例子,加深对逗号表达式和逗号分隔符的理解。

```
main()
{
    int a,b,c,d;
    c=(a=3,2*a);
    d=a=b=3,2*a;
```




```
printf("%d,%d,%d,%d\n",a,b,c,d);
}
```

程序的运行结果如图 5.2 所示。

在上面的代码中, 代码 `c=(a=3,2*a);` 是将一个逗号表达式的值赋给变量 `c`, 第一个表达式 `a=3`, 第二个表达式 `2*a`, 计算得到 6, 因此变量 `c` 的值是 6。

代码 `d=a=b=3,2*a;` 整个是一个逗号表达式, 这里逗号表达式的值同样是 `2*a` 的值, 结果是 6。但是变量 `d` 的值是在第一个表达式中计算得到的, 得到值为 3。

如果将代码改为下面的形式:

```
d=(a=b=3,2*a);
```

这样, 变量 `d` 得到的值就是逗号表达式的值, 即表达式 `2*a` 的值。运行结果如图 5.3 所示。

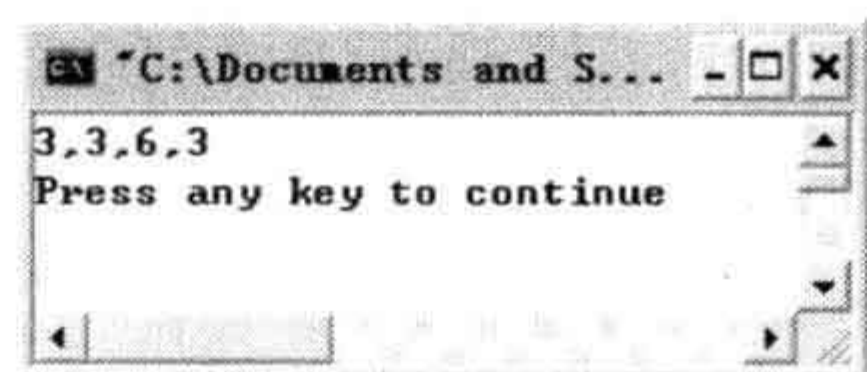


图 5.2 程序的运行结果

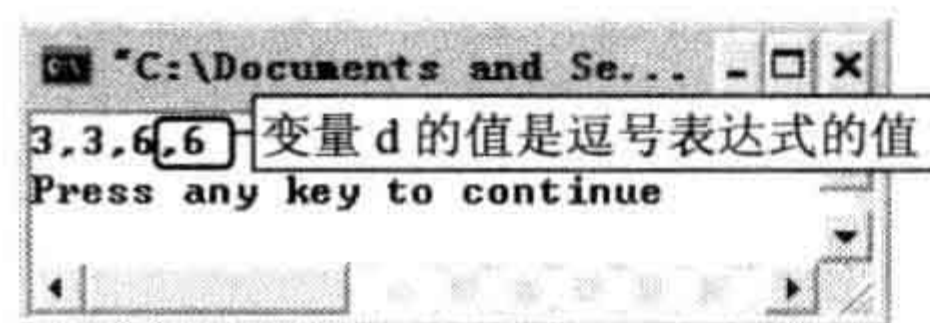


图 5.3 变量 `d` 的值变为 6

专家点评

在使用逗号运算符的时候, 要注意它的优先级和结合顺序。逗号运算符优先级最低, 并且是自左至右结合的。

问题 53 C 语言如何解释 `x=a+=b+c`?

问题阐述

C 语言如何解释 `x=a+=b+c` 这个表达式?

专家解答

复合的赋值运算符是 C 语言中特有的运算符。在赋值符 “=” 之前加上其他运算符, 可以构成复合的运算符。如 “+=”、“*=” 等。

由于赋值运算符是自右至左结合的, “=” 与 “+=” 优先级相同。

本题中, 表达式加上圆括号之后就是 `x=(a+=(b+c))`, 相当于 `x=a=a+(b+c)`。

对于本题中的表达式, 如果 `a=1`, `b=2`, `c=3`, 则有 `(b+c)` 的值为 5, `(a+=(b+c))` 的结果为 6, `a` 的值也为 6。因此, 最终 `x=6`, `a=6`。

专家点评

复合运算符能够简化程序和提高编译效率, 有利于编译, 能产生质量较高的目标代码。



Note



问题 54 $x=x+1$, $x+=1$, $x++$, 哪个效率最高?



Note

问题阐述

$x=x+1$, $x+=1$, $x++$, 这三个表达式哪个执行效率最高?

专家解答

第一个表达式 $x=x+1$ 的执行过程是先读取等号右边的 x 的地址, 计算 $x+1$ 的值, 然后读取等号左边的 x 的地址, 最后将等号右边的值传给等号左边的值。

第二个表达式 $x+=1$ 的执行过程是先读取等号右边的 x 的地址, 然后计算 $x+1$ 的值, 最后将得到的值传给左边的 x , 因为 x 的地址已在前面读出, 故省去了传值过程;

第三个表达式 $x++$ 的执行过程是先读取 x 的地址, 然后 x 自增 1;

因此, $x++$ 的效率最高。

专家点评

在编写程序的时候, 为了节省程序执行时间, 提高效率, 尽量要写效率高的代码。

问题 55 什么是运算符的目? 怎样进行区分?

问题阐述

运算符具有单目、双目和三目, 怎样区分运算符的目?

专家解答

不同的运算符要求有不同的运算对象个数, 如 “++” 和 “--” 为单目运算符, “+” 和 “-” 为双目运算符。因为 “++” 和 “--” 运算符为一元运算符, 只需要在运算符的一侧出现一个运算对象 (如 $i++$ 、 $--i$ 、 $*p$ 、 $\sim a$ 等), 所以 “++” 和 “--” 叫做单目运算符; 而 “+” 和 “-” 需要在运算符两侧各有一个运算对象, 也就是需要两个运算对象 (如 $1+2$ 、 $8-5$), 所以它们叫做双目运算符。由此可以看出, 运算符需要几个运算对象就叫几目运算符。

专家点评

条件运算符是 C 语言中唯一的一个三目运算符, 如 $c? a:b$ 。



问题 56 使用“++”和“--”运算符需要注意些什么?

问题阐述

“++”和“--”运算符经常被应用,使用这两种运算符需要注意些什么?

专家解答

在 C 语言中,“++”运算符和“--”运算符被经常应用,大家需要熟练掌握它们的用法,特别是在多个运算符混合使用的情况下,更应该注意使用。“++”表示自增 1,使变量值加 1;“--”表示自减 1,使变量值减 1。它们都是单目运算符,具有右结合性。使用的一般形式如下:

++i i 自增 1 后再使用 i;

--i i 自减 1 后再使用 i;

i++ 使用 i 后 i 的值再自增 1;

i-- 使用 i 后 i 的值再自减 1;

自增和自减运算符只适用于单个变量,而不能用于其他表达式。自增自减运算符的表达式一般形式为 $j=++i$ 和 $j=--i$ 。i 只能是变量,而不能用常量或表达式。如 $1++$ 或 $(a+b)++$ 都是不合法的。原因在于常量是不能改变的,而表达式没有存放值的地方。

自增自减运算符的结合性是自右向左的,例如, $-i++$, 若先计算 $-i$, 再算 $(-i)++$, 这是不合法的,因为 $-i$ 属于表达式。负号运算符和自增运算符的优先级相同,而结合性都是自右向左的。

专家点评

“++”和“--”运算符通常应用于循环语句中,使循环语句自动加 1 (或减 1),“++”运算符也常用于指针变量,使指针指向下一个地址。

问题 57 如何理解 $i+++j$?

问题阐述

C 语言系统如何处理表达式 $i+++j$?

专家解答

对于表达式 $i+++j$, 系统会默认为 $(i++)+j$ 来处理。表达式 $i+++j$ 是不规范的写法,如果在书写表达式的时候有 3 个连续的运算符 (“+” 或 “-”), 应该使用空格符来进行分隔。如果不加分隔,编译系统一般会按尽量取大的原则处理。所以,对于表达式 $i+++j$,



Note



如果想按照 $(i++) + j$ 运算, 标准的写法应该为 $i++ + j$ 。

在书写多个连续运算符的表达式的时候, 要尽量使用分隔符或者圆括号进行分隔, 这样即能使程序脉络清晰, 容易阅读, 又能避免混淆, 产生错误。



Note

专家点评

书写代码的时候, 应该使用空格对表达式进行分隔, 以使代码可读性更高, 提高程序的质量。

问题 58 赋值表达式中什么是左值和右值? 数组名作为左右值时又具有怎样的意义?

问题阐述

赋值表达式中可以分为左值和右值, 那么什么是左值和右值? 数组名做为左右值时又具有怎样的意义?

专家解答

左值是指可以被赋值的表达式, 也就是赋值符号左侧的表达式。由此可以知道, 右值就是指出现在赋值符右侧的表达式。每一个赋值语句都有一个左值和一个右值。

(1) 左值必须是变量。左值必须是内存中一个可存储的变量, 而不能是一个常量或者表达式。例如, 下面是正确的左值。

```
int i;  
int *p;  
i=5;  
*p=3;
```

i 是一个整型变量, 在内存中有一个对应的存储位置。因此, 语句 $i=5$ 中, i 可以作为一个左值。在语句 $*p=3$ 中, $*p$ 表示 p 指向的内存区域, 因此, $*p$ 是一个左值。而下面几个例子就不是左值。

```
#define PRICE 10  
int i,j;  
PRICE=20;  
(i+j)=10;
```

上面的代码中, $PRICE$ 是一个常量值, 其值不能改变, 因为常量不表示内存中可存储的位置; 而 $(i+j)$ 是一个表达式, 也不能表示内存中可存储的一个位置。所以这两个赋值语句的左值都是不正确的。

(2) 右值可以是常量或者表达式。右值可以是常量或者表达式, 例如:



```
#define PRICE 10
int i,j;
int *p;
i=5;
j=i+5
*p=PRICE;
```



Note

(3) 赋值语句必须有一个左值和一个右值。一条赋值语句必须有一个左值和一个右值，否则将无法通过编译。

数组名作为左右值的意义：

当数组名作为左值时，是错误的。编译器会认为数组名作为左值代表的意思是数组的首元素的首地址，但是这个地址开始的一块内存是一个整体，只能访问数组中某个元素，而无法访问整个数组，所以可以把 `a[3]` 当左值，而不能把 `a` 当左值。

数组名代表数组的首地址，所以数组名作为右值时将数组的首地址赋给赋值符左侧的变量。

专家点评

数组名不可以作为左值，而数组中的元素是可以作为左值的。因为数组中的每个元素也可以被看作是一个变量，其在内存中有对应的存储位置。

问题 59 如何确定条件表达式的结果的数据类型？

问题阐述

条件表达式是三目的表达式，由 3 个运算对象组成，那么如何确定表达式最后结果的数据类型呢？

专家解答

条件表达式是由条件运算符连接起来的三目表达式，即由 3 个运算对象组成，这个表达式结果的数据类型是由冒号前后两个表达式中类型高的那一个类型决定的。

例如，下面的代码。

```
#include<stdio.h>
int main()
{
    int x=10,y=2,a;
    a=sizeof(x>y?x:y+0.1);
    printf("%d\n",a);
}
```

上面的代码最后输出的结果为 8，由此可见，表达式的结果的数据类型为 `double` 类型，



与冒号后的表达式 $y+0.1$ 的数据类型相同，但是表达式的值是冒号前 x 的值。

专家点评

使用条件表达式可以将 `if...else...` 语句简化，以达到对程序代码简化的目的。



Note

问题 60 “%” 运算符是否可以对小数进行运算？

问题阐述

如题，“%” 运算符是否可以对小数进行运算？

专家解答

“%” 运算符，称为求余运算符或者模运算符，要求 “%” 两侧都为整型数据，否则将会产生错误。

例如：

```
#include<stdio.h>
main()
{
    float i,j;
    int k;
    printf("please input:\n");
    scanf("%f,%f",&i,&j);
    k=i%j;                      /*此处错误*/
    printf("%d",k);
}
```

运行该程序，会提示如图 5.4 所示的错误。

```
Compiling E:\CPRO\24.C:
•Error E:\CPRO\24.C 8: Illegal use of floating point in function main
```

图 5.4 编译错误信息

需要将程序中的如下语句。

```
k=i%j;
```

改写成：

```
k=(int)i%(int)j;
```

专家点评

如果定义的变量不是整数类型，可以使用强制类型转换方式将其转换为整型数据，再进行计算。



问题 61 “/” 运算符得到的结果一定为整数吗?

问题阐述

“/” 运算符得到的结果一定为整数吗?

专家解答

C 语言中的除法运算符“/”有着比较灵活的规定。两个整数相除,结果为整数,运算中的两个数有一个数为实数,则结果是 double 类型,因为所有实数都按照 double 型进行运算。做整数除法的时候舍去小数部分,但是如果除数或者被除数中有一个为负值,则舍入的方向是不固定的,多数机器采取“向零取整”的方式进行运算。如 $5/3=1$, $-5/3=-1$, 向零取整。

例如,下面的代码。

```
#include <stdio.h>
void main()
{
    double i;                /*声明 double 型变量*/
    i=9.5+1/2;               /*为变量赋值*/
    printf("%f",i);          /*输出结果*/
}
```

这里进行的是实数运算,本想将 9.5 与 $1/2$ 计算的结果 0.5 相加,得到数值 10。可是事与愿违,结果为 9.5。运行结果如图 5.5 所示。

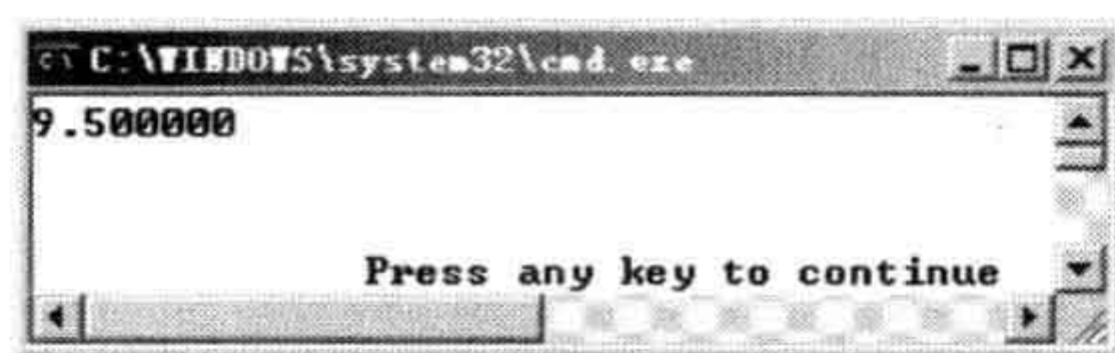


图 5.5 运行结果

使用除法运算符“/”对两个整数进行运算的时候,得到的结果也是一个整数。 $1/2$ 得到的结果去掉小数部分得到的整数为 0。所以,上面的运算结果不是预期的结果。

将 $1/2$ 改为 $1.0/2$ 即可得到一个实数的结果。将除数和被除数中的任意一个数设置为实型,都会得到一个双精度的结果。代码如下。

```
double i;
i=9.5+1.0/2;
printf("%f",i);
```

专家点评

使用运算符“/”进行运算的时候,要注意表达式中运算对象的数据类型,以免因为数据类型疏忽而导致结果的误差。



Note



Note

问题 62 在进行多种数据类型混合运算的时候，数据类型自动转换有哪些规则？

问题阐述

如题，在进行多种数据类型混合运算的时候，数据类型什么时候会自动转换？有哪些规则？

专家解答

自动转换发生在不同数据类型的量混合运算时，由编译系统自动完成。自动转换遵循以下规则：

- (1) 若参与运算量的类型不同，则先转换成同一类型，然后进行运算。
- (2) 转换按数据长度增加的方向进行，以保证精度不降低。如 int 型和 long 型运算时，先把 int 量转换成 long 型后再进行运算。
- (3) 所有的浮点运算都是以双精度进行的，即使仅含 float 单精度量运算的表达式，也要先转换成 double 型，再作运算。
- (4) char 型和 short 型参与运算时，必须先转换成 int 型。
- (5) 在赋值运算中，赋值号两边量的数据类型不同时，赋值号右边量的类型将转换为左边量的类型。如果右边量的数据类型长度比左边长时，将丢失一部分数据，这样会降低精度，丢失的部分将按四舍五入向前舍入。

当有如下定义。

```
char a;  
int b;  
long int c;  
float d;  
double e;  
result=(a+b)*(c-a)/(d/e);
```

则其转换关系如图 5.6 所示。

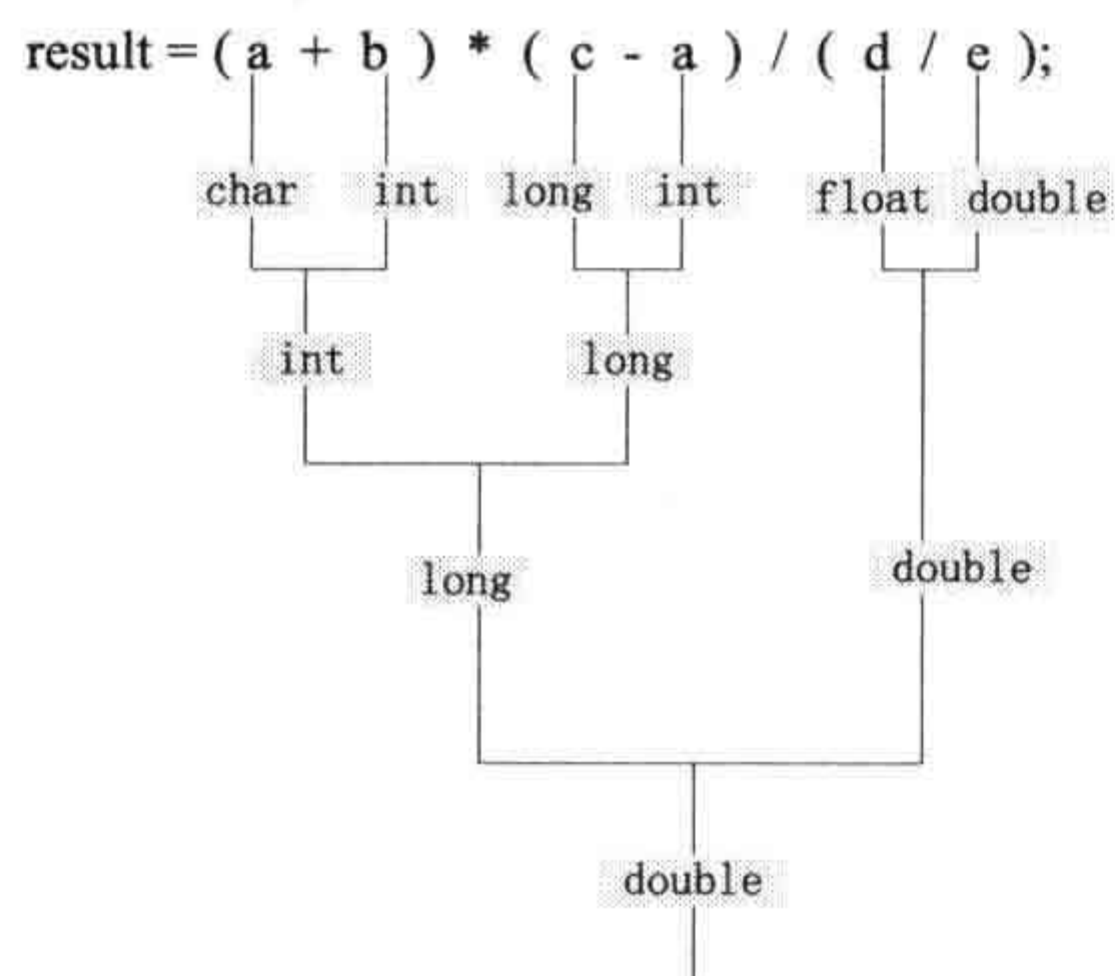


图 5.6 自动类型转换关系



专家点评

大家要了解这些自动转换规则，并在实际开发中进行熟悉，以使程序的运行结果准确无误。

问题 63 C 语言中有哪些简化的运算表达式？



Note

问题阐述

C 语言是一种简单的程序设计语言，那么 C 语言中有哪些简化的运算表达式？

专家解答

C 语言中简化的运算表达式总结如下：

- (1) 加 1 运算。当一个变量要执行加 1 运算时，如 $a=a+1$ ，可以写成 $a++$ 的形式。
- (2) 减 1 运算。当一个变量要执行减 1 运算时，如 $a=a-1$ ，可以写成 $a--$ 的形式。
- (3) 加一个常量的运算。当一个变量与一个常量相加时，如 $a=a+3$ ，可以写成 $a+=3$ 的形式。
- (4) 减一个常量的运算。当一个变量与一个常量相减时，如 $a=a-3$ ，可以写成 $a-=3$ 的形式。
- (5) 乘一个常量的运算。当一个变量与一个常量相乘时，如 $a=a*3$ ，可以写成 $a*=3$ 的形式。
- (6) 除以一个常量的运算。当一个变量与一个常量相除时，如 $a=a/3$ ，可以写成 $a/=3$ 的形式。

专家点评

这些简化的表达式可以达到精简代码、增加程序编译速度的作用。

问题 64 使用逻辑表达式需要注意哪几点问题？

问题阐述

逻辑表达式在程序设计时经常被应用，使用逻辑表达式需要注意哪几点问题？

专家解答

逻辑运算符两侧的操作数除了可以是 0 和非 0 的整数以外，也可以是其他任何类型的数据，如，实型、字符型等。

在计算逻辑表达式时，只有在必须执行下一个表达式才能求解时，才求解该表达式。



也就是说，并不是所有的表达式都被求解，它有两种情况：

(1) 对于逻辑与运算，如果第一个操作数被判定为“假”，则系统不再判定或求解第二个操作数。

例如：

$1 > 3 \&\& 1 > 0$ ，由于 $1 > 3$ 为假，相与的结果也就为假，后面的 $1 > 0$ 就不会执行。只有前项为真时，才执行后项。

(2) 对于逻辑或运算，如果第一个操作数被判定为“真”，系统不再判定或求解第二个操作数。

例如：

$3 > 1 \|\| 3 > 5$ ，由于 $3 > 1$ 为真，相或的结果也就为真，后面的 $3 > 5$ 就不会执行。只有前项为假时，才执行后项。

逻辑运算符（ $\&\&$ 、 $\|\|$ 、 $!$ ）是将操作数当成非真即假，要么非假即真的情况。通常都是将 0 当成假，非 0 则为真。

专家点评

要注意逻辑表达式和位运算表达式的使用，不要对这两种表达式产生混淆。

问题 65 $i++*i++$ 这样的表达式为什么不能得到预期的结果？

问题阐述

定义 $i=5$ ； $i++*i++$ 的结果为什么是 25，而不是 30？

专家解答

大家都知道，后缀自增和后缀自减运算符（即 $i++$ 和 $i--$ ）是在输出值之后进行自增自减运算，但是不能确保在任何情况下自增和自减运算在输出变量原值之后立即执行变量的自增或自减，也不能确定变量的更新会在此表达式执行之后的某个时刻进行。在本问题中，编译器选择使用变量的原值相乘之后，再对二者进行自增运算。然而，看一下下面的代码。

```
#include<stdio.h>
int main()
{
    int i=5;
    printf("%d,%d\n",i++*i++,i);
    printf("%d\n",i);
}
```




上面代码运行的结果如图 5.7 所示。

也就是说，当第一个 `printf` 语句执行完成之后，变量值才进行自增。

在同一个表达式中，使用导致同一对象修改两次或者修改以后有被引用的自增自减和赋值操作符的任何组合，总是被认为未定义的。这是不规范的写法。

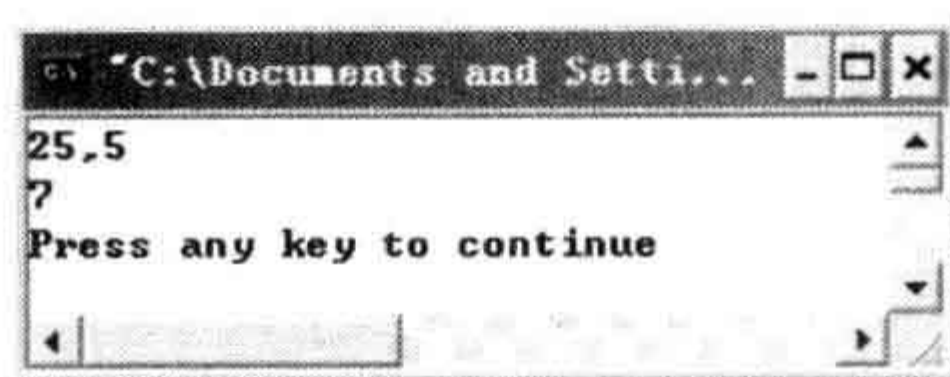


图 5.7 程序运行结果



Note

专家点评

当对语句中的变量使用自增或自减运算符时，该变量不应在语句中出现一次以上，因为求值的顺序取决于编译器。编写代码时，不要对顺序作假设，也不要编写在某一机器上能够如期运行但没有明确定义行为的代码。这是比较容易出错的地方，应避免这样的错误。

问题 66 `a[i]=i++;`这样的代码正确吗？

问题阐述

`a[i]=i++;`这样的代码正确吗？

专家解答

语句 `a[i]=i++;`是否能正常执行呢？答案很显然是不能的。因为子表达式 `i++` 有一个副作用，就是它会改变 `i` 的值，由于 `i` 在同一个表达式的其他地方被引用，这将导致一个无法判断的结果，无从判断 `a[i]` 中的 `i` 是旧值还是新值。在使用中应该避免此类现象的出现。

专家点评

本问题可与上一问题归结为一个问题。在编写程序时，要尽量避免书写这种不规范、容易混淆的代码。

问题 67 编写表达式时需要注意什么？

问题阐述

在编写表达式时，应该注意什么？

专家解答

C 语言中，表达式是用运算符将各个操作对象连接成符合 C 语言语法规则的运算式子，通常由变量、常量、运算符和函数组成。



Note

在书写表达式时，应注意以下事项：

- (1) 注意组成表达式的各操作对象的值；
- (2) 如果表达式具有变量，注意变量类型，以确定表达式结果类型；
- (3) 确定组成表达式的各种运算符的功能；
- (4) 清楚表达式的运算顺序，以确定表达式得到的预期结果。

专家点评

为了得到正确的计算结果，应该在编写表达式的时候注意这些事项，以免由于表达式编写疏忽而产生程序结果的错误。

问题 68 如何理解 $c=a,b;?$

问题阐述

对于表达式 $c=a,b;$ 和 $d=(a,b);$ 该如何进行理解？它们的值都是怎样的？

专家解答

在 C 语言中，逗号有两个作用，一是用来分隔函数参数，二是作为逗号运算符。本题主要考虑的是逗号运算符，根据逗号运算符的规则，将最后表达式的值作为逗号表达式的值。如下面的代码。

```
main()
{
    int a, b, c, d;
    a=3;
    b=5;
    c=a,b;
    d=(a,b);
    printf("c=%d", c);
    printf("d=%d", d);
}
```

粗心的读者会认为 c 的值应为 5，但答案却是错误的。这是什么原因呢？在 C 语言中逗号运算符的优先级最低，应被最后考虑。

这里，表达式 $c=a,b;$ 是一个逗号表达式，其中包括一个赋值表达式，先进行赋值运算， c 的值是 3 而不是 5，但整个逗号表达式的值是 b （即 5）。这也往往是许多人忽略的地方所在，应尽量避免此种情况的发生。表达式 $d=(a,b);$ 是一个赋值表达式，将一个逗号表达式的值赋给 d ，故 d 的值为 b （即 5）。



专家点评

对于表达式，既要考虑运算符的特点，也要考虑运算符之间的优先级顺序，不能粗心大意而产生错误。

问题 69 为无符号类型变量赋值时，数据类型应怎样转换？



Note

问题阐述

为无符号类型变量赋值时，数据类型应怎样转换？

专家解答

首先看下面的代码。

```
int main (void)
{
    unsigned int a = 6;
    int b = -20;
    (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

上面代码中涉及了条件运算符。条件运算符是 C 语言中唯一一个三目运算符，其规则为：在计算第一个表达式的值之后，若值为真，则将第二个表达式的结果作为整个表达式的值；若第一个表达式的值为假，则将第三个表达式的结果作为整个表达式的值。

上面代码输出的是“>6”，因为当表达式中存在有符号类型和无符号类型时，所有的操作数都自动转换为无符号类型。因此-20 变成了一个非常大的正整数，所以该表达式计算出的结果大于 6。

专家点评

在 C 语言中，无符号类型只能存放不带符号的整数，不能存放负数。当为其赋值为负数时，会自动转换为无符号类型数值，其取值范围是 0~65535。

问题 70 C 语言表达式的求值顺序总是按照运算符的结合性保证“自左至右”或者“自右至左”吗？

问题阐述

C 语言表达式的求值顺序是怎样的？总是保证“自左至右”或者“自右至左”吗？



专家解答



Note

一般情况下，C 语言中运算符的结合性有“自左至右”和“自右至左”两种。但是，C 语言并不是总是按照这个顺序去求值的。因为运算符的优先级和括弧只能赋予表达式的计算顺序，而对于函数来说，就不能确定执行顺序了。

一般 C 语言总是首先求函数值，再求复杂表达式的值，最后求简单表达式的值。另外，C 语言编译器为了进一步优化代码，常会改变表达式的求值顺序。如 $b+c*d+f()+g()*h()$ ，尽管乘法运算在加法运算之前，但是这并不能说明这三个函数哪个将会被先执行。

专家点评

在书写代码时，最好明确指定运算符的优先级。这样，能够确保表达式被正确求值，而且编译器不会为了优化代码而重新安排运算符优先级。

第 6 章

输入/输出函数

- ▶▶ 函数 printf() 的基本格式是什么?
- ▶▶ 如何认识 printf() 函数的格式字符?
- ▶▶ 函数 printf() 的标志有几种? 如何使用?
- ▶▶ 如何控制输出最小宽度?
- ▶▶ 如何输出精度?
- ▶▶ 如何控制长度?
- ▶▶ 如何动态设置输出宽度和精度?
- ▶▶ printf() 函数的返回值是什么?
- ▶▶ 如何理解输出列表?
- ▶▶ 函数 scanf() 的基本格式是什么?
- ▶▶ scanf() 函数的格式字符是什么?
- ▶▶ scanf() 函数应注意的问题是什么?
- ▶▶ scanf() 函数的返回值是什么?
- ▶▶ 如何使用 getchar() 函数?
- ▶▶ getch() 函数如何使用?
- ▶▶ 如何应用 gets() 函数?
- ▶▶ 如何应用 putchar() 函数?
- ▶▶ puts() 函数该如何应用?
- ▶▶ 如何控制多数值的输入?
- ▶▶ 如何输入字符数组?



问题 71 函数 printf() 的基本格式是什么?



问题阐述

Note

在前面的章节中,大家都习惯了使用 printf() 函数进行输出,但是也只是使用,那么其基本格式又是怎样的呢?

专家解答

printf() 函数的作用是向终端输出若干个任意类型的数据。该函数是一个标准库函数,其函数原型在头文件 stdio.h 中。printf() 函数调用的一般形式如下。

printf(格式控制, 输出列表)

参数说明:

- ☑ 格式控制: 用双引号括起来的字符串, 也称“转换控制字符串”。“格式控制”分为两种, 即格式字符串和非格式字符串。格式字符串是以“%”开头的字符串, 在“%”后面跟有各种格式字符, 以说明输出数据的类型、形式、长度、小数位数等; 非格式字符串也就是通常所说的普通字符, 即在输出时原样输出。例如:

printf("%d,%d",a,b);

这里的“%d”就是格式字符串;“,”是非格式字符串,即普通字符;a和b是输出列表。

- ☑ 输出列表: 需要输出到屏幕的数据, 可以是常量、变量或表达式。若为表达式, 则将输出表达式的值。

下面举例说明, 代码如下。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a=8;
    float f=0.892;
    printf("这是一个特殊的输出! \n");
    printf("整型变量的输出是 a=%d\t, 实型变量的输出是: f=%f\n",a,f);
    system("PAUSE");
    return 0;
}
```

上述程序的运行结果如图 6.1 所示。

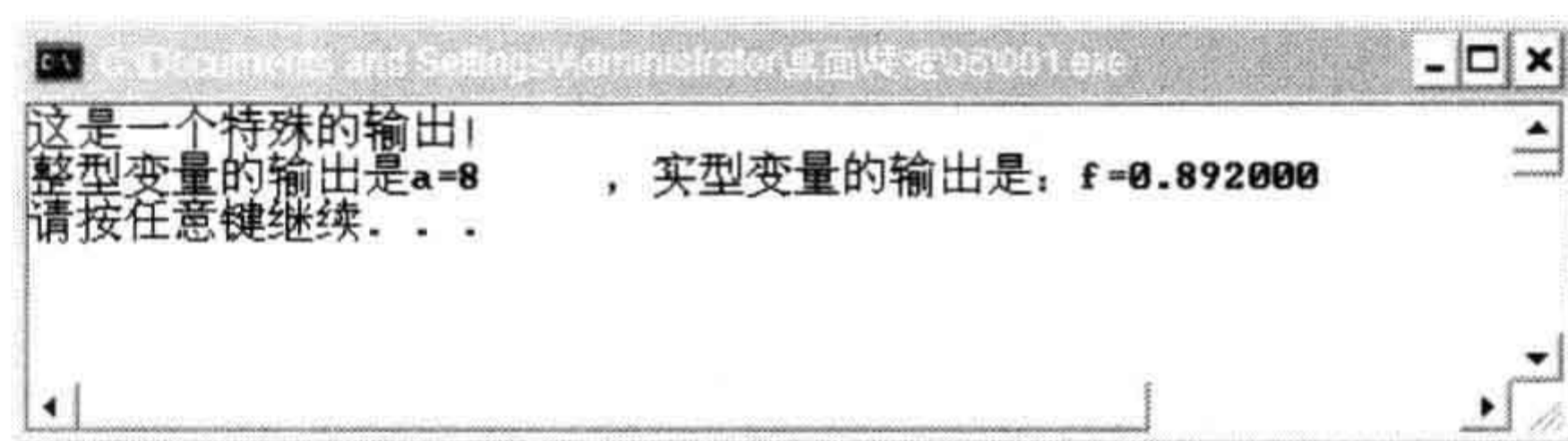


图 6.1 演示程序输出结果

在上面的程序中，第一个输出 `printf()` 函数中只有格式控制字符串，而没有输出列表，所以只是输出了一串文字信息。最后的“`\n`”是一个转义字符，用于换行，即后面的输出函数将在新的一行输出。

第二个输出函数 `printf()` 中包含两部分参数，一部分参数是提示文字、控制字符以及转义字符；另一部分参数则是输出列表，输出两个变量，用逗号隔开。该函数的输出内容如图 6.1 中第二行所示。

专家点评

格式控制字符串中的每个格式字符，都需要在输出列表中有一个表达式与之相对应，否则将输出一个不确定的数值。

问题 72 如何认识 `printf()` 函数的格式字符？

问题阐述

前面介绍了 `printf()` 函数的基本格式，那么对输出格式到底该如何控制呢？

专家解答

对不同类型的数据用不同的格式字符串，其中常用的有以下几种格式字符。

(1) 格式字符 `d`。格式字符 `d` 用于输出十进制整数。有以下几种用法：

- ☒ `%d`：按整型数据的实际长度输出。
- ☒ `%md`：`m` 为指定的输出字段的宽度。如果数据的位数小于 `m`，则左端补以空格；若大于 `m`，则按实际位数输出。
- ☒ `%ld`：输出长整型数据。

下面举例说明格式字符的具体应用代码如下。

```
#include <stdio.h>
main()
{
    int i, k;
    long j;
    i = 36;
```

**Note**



Note

```

j = 42767;
k = 123;
printf("%d,%d\n", i, k);           /*以十进制形式输出 i 和 k*/
printf("%4d,%2d\n", i, k);        /*以指定格式输出*/
printf("%ld", j);
}

```

程序运行结果如图 6.2 所示。

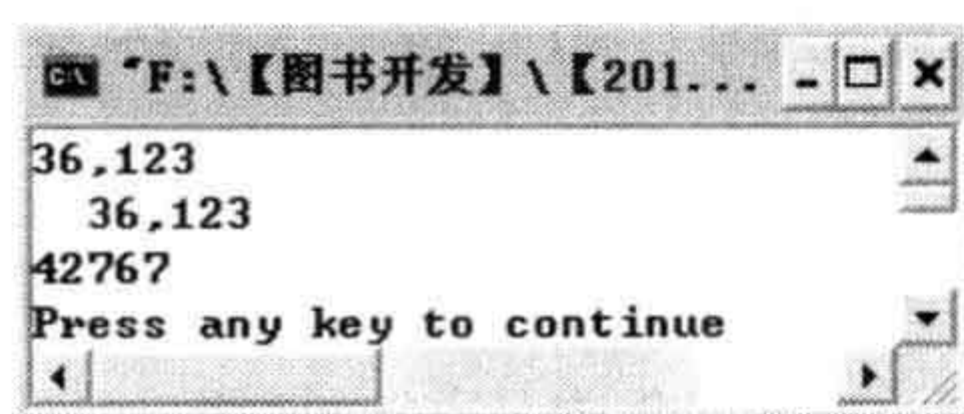


图 6.2 十进制输出

(2) 格式字符 o。格式字符 o 用于以八进制形式输出整数。有以下几种用法:

- ☑ %o: 按整型数据的实际长度输出。
- ☑ %mo: m 为指定的输出字段的宽度。如果数据的位数小于 m, 则左端补以空格; 若大于 m, 则按实际位数输出。
- ☑ %lo: 输出长整型数据。

下面举例说明格式字符 o 的具体应用, 代码如下。

```

#include<stdio.h>
main()
{
    int i=32;
    printf("%d,%o",i,i);           /*以八进制形式输出*/
}

```

程序运行结果如图 6.3 所示。

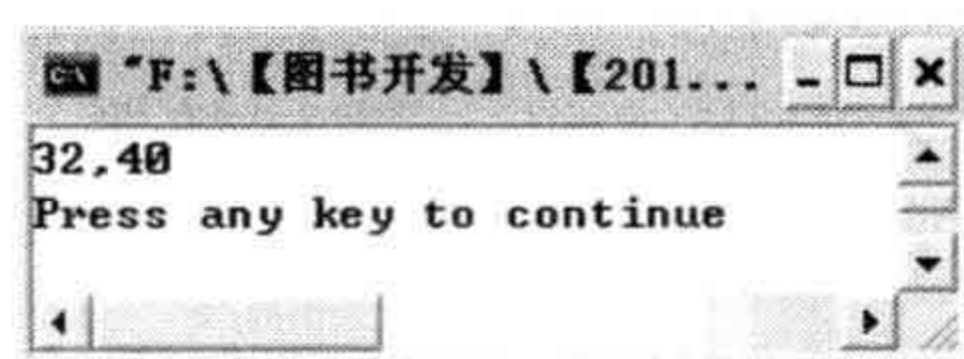


图 6.3 八进制输出

注意:

由于八进制数不带符号, 故将内存单元中各位转换成八进制时将符号位也一起作为八进制数的一部分输出。

(3) 格式字符 x。格式字符 x 用于以十六进制形式输出整数。有以下几种用法:

- ☑ %x: 按整型数据的实际长度输出。
- ☑ %mx: m 为指定的输出字段的宽度。如果数据的位数小于 m, 则左端补以空格, 若大于 m, 则按实际位数输出。



☑ %lx: 输出长整型数据。

下面举例说明格式字符 x 的具体应用, 代码如下。

```
#include<stdio.h>
main()
{
    int i=32;
    printf("%d,%o,%x",i,i,i);          /*分别以十进制、八进制、十六进制输出数据*/
}
```

程序运行结果如图 6.4 所示。

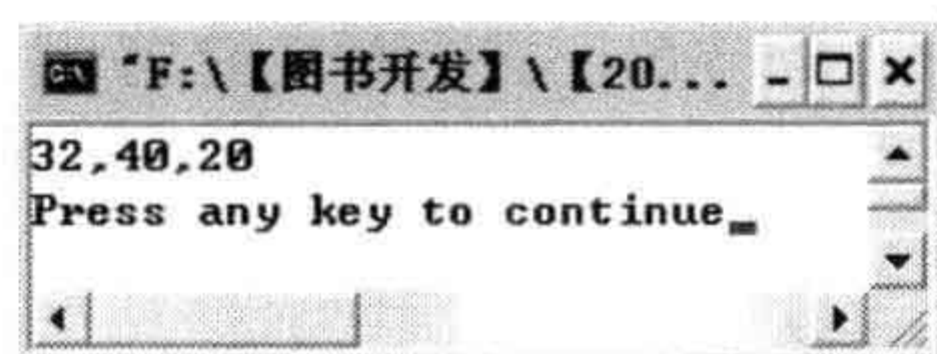


图 6.4 十六进制输出

(4) 格式字符 u。格式字符 u 用于以十进制形式输出无符号型数据。
下面举例说明格式字符 u 的具体应用, 代码如下。

```
#include<stdio.h>
main()
{
    int i,j;
    i=-5,j=5;                          /*为变量赋初值*/
    printf("%d,%d\n",i,j);
    printf("%u,%u\n",i,j);             /*以无符号形式输出 i 和 j*/
}
```

程序运行结果如图 6.5 所示。

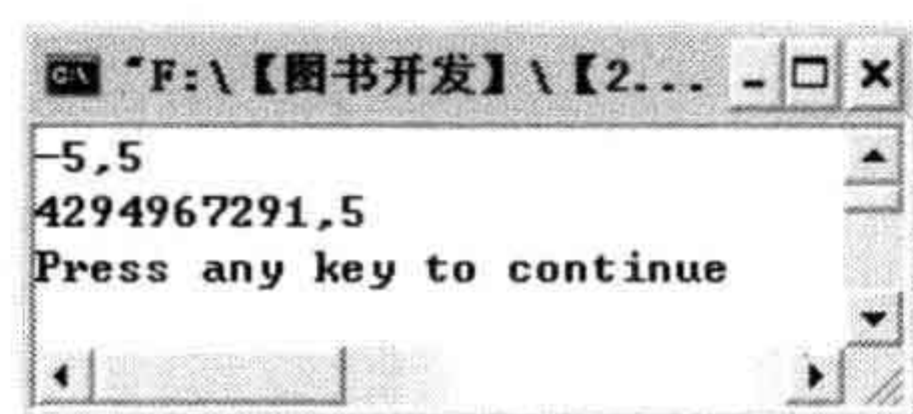


图 6.5 无符号形式输出

(5) 格式字符 c。格式字符 c 用于输出一个字符。

说明:

0~255 范围内的整数, 也可以用字符形式输出。同样, 一个字符数据也可以用整数形式输出。

下面举例说明格式字符 c 的具体应用, 代码如下。

```
#include<stdio.h>
main()
```



Note



Note

```

{
    int i=99;
    char ch='a';
    printf("%d,%c\n",ch,i);
}

```

/*以十进制和字符形式输出*/

程序运行结果如图 6.6 所示。

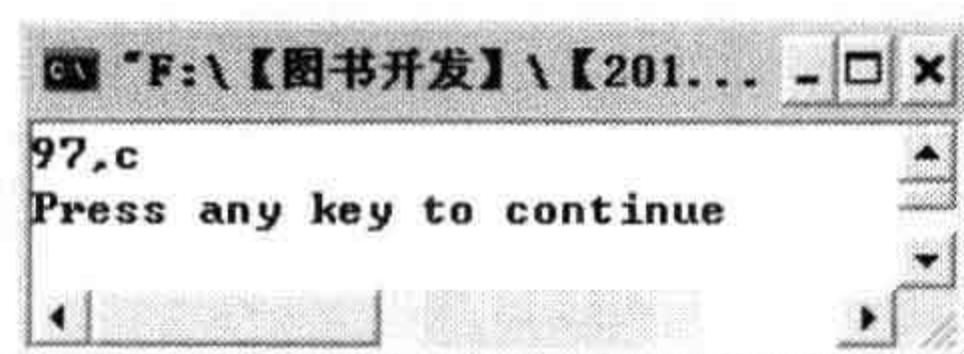


图 6.6 字符形式输出

(6) 格式字符 s。格式字符 s 用于输出一个字符串。有以下几种用法：

- ☑ %s: 将字符串按实际长度输出。
- ☑ %ms: 输出的字符串占 m 列。如果字符串本身长度大于 m, 则突破 m 的限制, 将字符串全部输出; 若字符串长度小于 m, 则左补空格。
- ☑ %-ms: 如果字符串长度小于 m, 则在 m 列范围内, 字符串向左靠, 右补空格。
- ☑ %m.ns: 输出占 m 列, 但只取字符串中左端 n 个字符。这 n 个字符输出在 m 列的右侧, 左补空格。
- ☑ %-m.ns: 输出长整型数据。输出占 m 列, 但只取字符串中左端 n 个字符。这 n 个字符输出在 m 列的左侧, 右补空格。

注意:

如果以 %m.ns 或 %-m.ns 形式输出时 m 的值小于 n, 则 m 自动取 n 值。

下面举例说明格式字符 s 的具体应用, 代码如下。

```

#include<stdio.h>
main()
{
    char *str="helloworld";
    printf("%s\n%10.5s\n%-10.2s\n%.3s",str,str,str,str);
}

```

/*以指定格式输出*/

程序运行结果如图 6.7 所示。

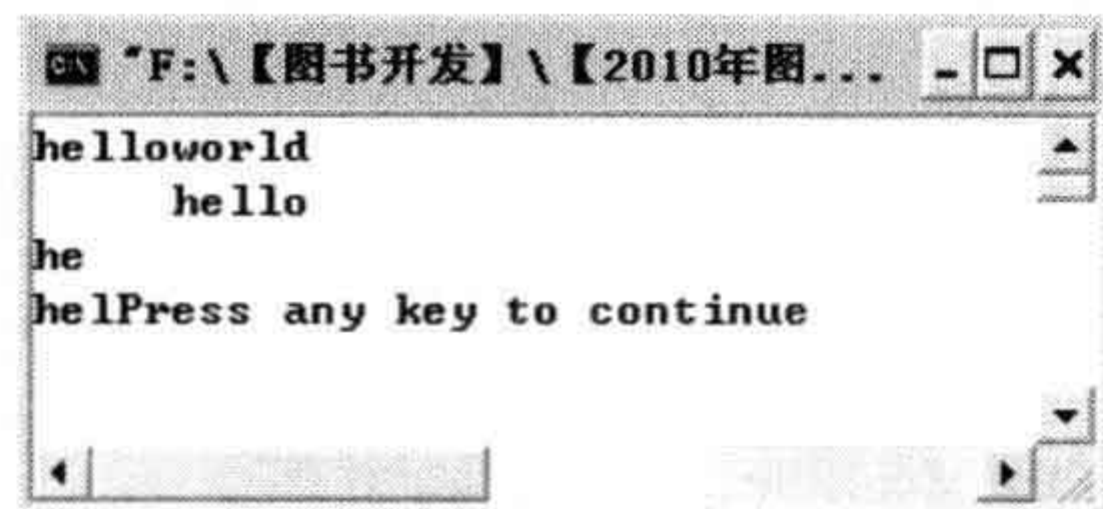


图 6.7 字符串输出



(7) 格式字符 f。格式字符 f 用于以小数形式输出实型数据。有以下几种用法:

- ☑ %f: 不指定字段宽度, 整数部分全部输出, 小数部分输出 6 位。
- ☑ %m.nf: 输出的数据占 m 列, 其中有 n 位小数。如果数值长度小于 m, 则左端补空格。
- ☑ %-m.nf: 输出的数据占 m 列, 其中有 n 位小数。如果数值长度小于 m, 则右端补空格。



Note

注意:

以%f形式输出的数据并不全都是准确的, 只有前7位数字是有效数字。双精度数同样可用%f输出。

下面举例说明格式字符 f 的具体应用, 代码如下。

```
#include<stdio.h>
main()
{
    float i=2998.453257845;          /*定义单精度型并赋初值*/
    double j=2998.453257845;         /*定义双精度型并赋初值*/
    printf("%f\n%15.2f\n%-10.3f\n%f\n",i,i,i,j); /*以指定的格式输出 i 和 j*/
}
```

实型数据输出, 程序运行结果如图 6.8 所示。

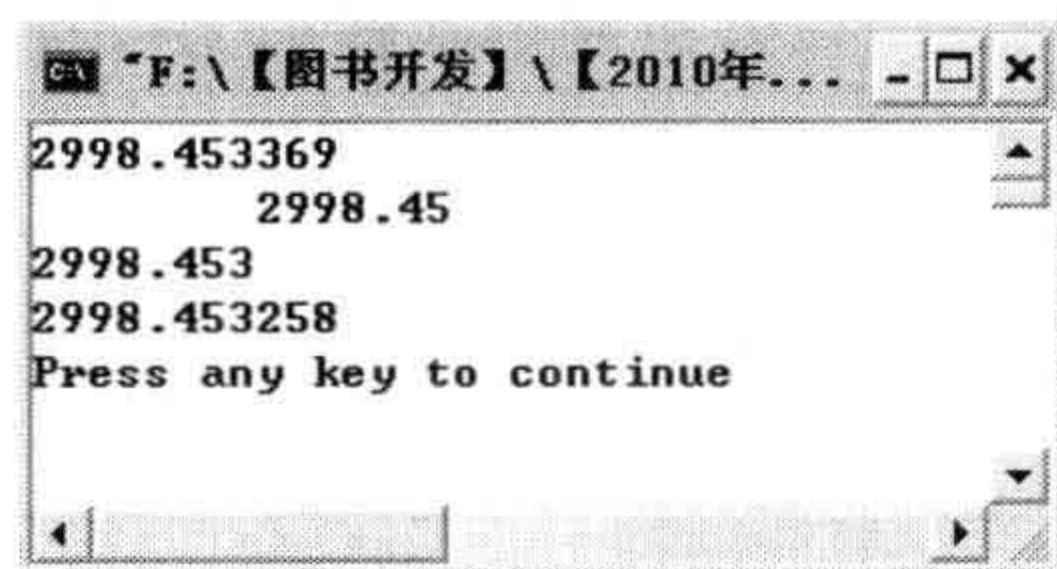


图 6.8 以小数形式输出实型数据

(8) 格式字符 e。格式字符 e 用于以指数形式输出实型数据。有以下几种用法:

- ☑ %e: 不指定输出数据所占的宽度和小数位数。
- ☑ %m.ne: 输出的数据占 m 位, 其中有 n 位小数。如果数值长度小于 m, 则左端补空格。
- ☑ %-m.ne: 输出的数据占 m 位, 其中有 n 位小数。如果数值长度小于 m, 则右端补空格。

注意:

输出的指数形式中的指数符号“+”算一位。

下面举例说明格式字符 e 的具体应用, 代码如下。

```
#include<stdio.h>
```




Note

```
main()
{
    float i=2998.453257845;
    double j=2998.453257845;
    printf("%e\n%15.2e\n%-10.3e\n",i,i,j);          /*以指数形式*/
}
```

程序运行结果如图 6.9 所示。

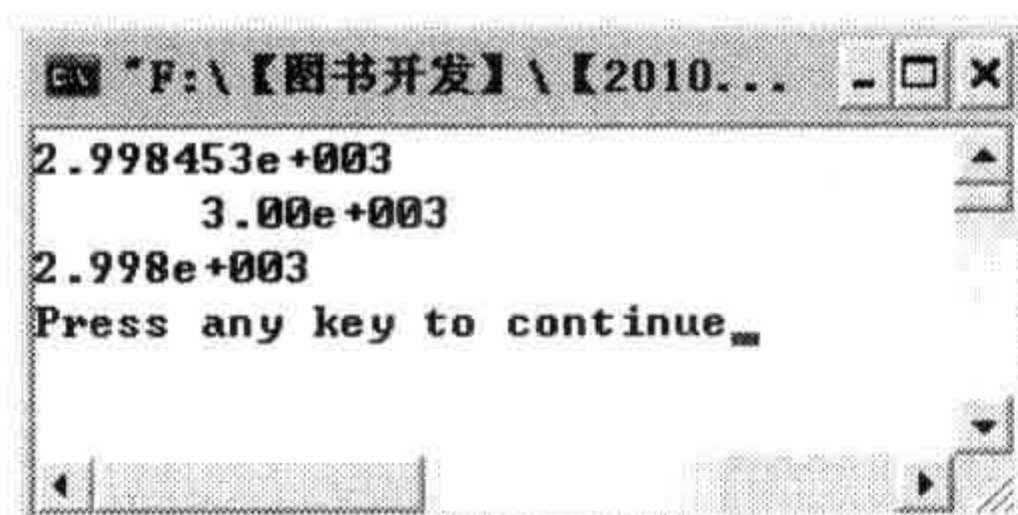


图 6.9 指数形式输出

(9) 格式字符 g。格式字符 g 用于以%f%e 中较短的输出宽度输出单、双精度实数，即根据数字的大小，自动选择 f 格式或 e 格式。

将以上介绍的几种格式字符进行归纳，如表 6.1 所示。

表 6.1 printf()函数的格式字符

表示输出类型的格式字符	含 义
d	以十进制形式输出带符号整数（正数不输出符号）
o	以八进制形式输出无符号整数（不输出前缀O）
x	以十六进制形式输出无符号整数（不输出前缀OX）
u	以十进制形式输出无符号整数
c	输出单个字符
s	输出字符串
f	以小数形式输出单、双精度实数
e	以指数形式输出单、双精度实数
g	以%f%e中较短的输出宽度输出单、双精度实数

说明：

(1) 除了格式字符 x、e、g 在使用时可以大写外，其余格式字符必须小写。

(2) 在进行字符“%”的输出时要注意，必须在格式控制中连写两个%，才能得到预期的效果。例如：

```
printf("%d%%",100);
```

输出为：100%。



问题 73 函数 printf() 的标志有几种? 如何使用?

问题阐述

通过前面的介绍,我们对 printf() 函数的格式字符已有所认识,但不清楚其前面的标志都有哪些。又该如何使用呢?

专家解答

在 printf() 函数中,格式字符前面的标志主要包括 5 种,如表 6.2 所示。

表 6.2 printf() 函数的标志

标 志	含 义
-	输出项左对齐,右边空格补齐
+	输出项的正负号
(空格)	输出数据是正数时,输出空格;负数输出负号
#	对所有实数,#保证即使不跟任何数字,也输出一个小数点
0	对所有数字格式,用前导零而不是使用空格填充字段宽度。如果有-标志或指定了精度,则可以忽略本标志

专家点评

标志的使用就是为了使输出项更加美观。读者可以灵活使用,以达到输出的美观性。

问题 74 如何控制输出最小宽度?

问题阐述

上面谈到了数据的美观性问题,这不仅需要使用标志进行占位,还需要对宽度等进行控制。那么如何控制宽度呢?

专家解答

控制宽度的问题,处理起来其实很简单。如果输出数据的实际位数大于定义的宽度,则按实际位数输出;如小于定义的宽度,则以空格或 0 补齐;也可以根据自定义的标志来输出。具体使用方式参见如下示例程序。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
```



Note



Note

```
int i=-89,j=890201;
printf("%d%d\n",i,j);
printf("%4d%4d\n",i,j);
printf("%+4d%+4d\n",i,j);
printf("%#x%#X\n",i,j);
printf("%04d%04d\n",i,j);
printf("% d% d\n",i,j);
printf("%-4d%-4d\n",i,j);
system("PAUSE");
return 0;
}
```

程序运行结果如图 6.10 所示。

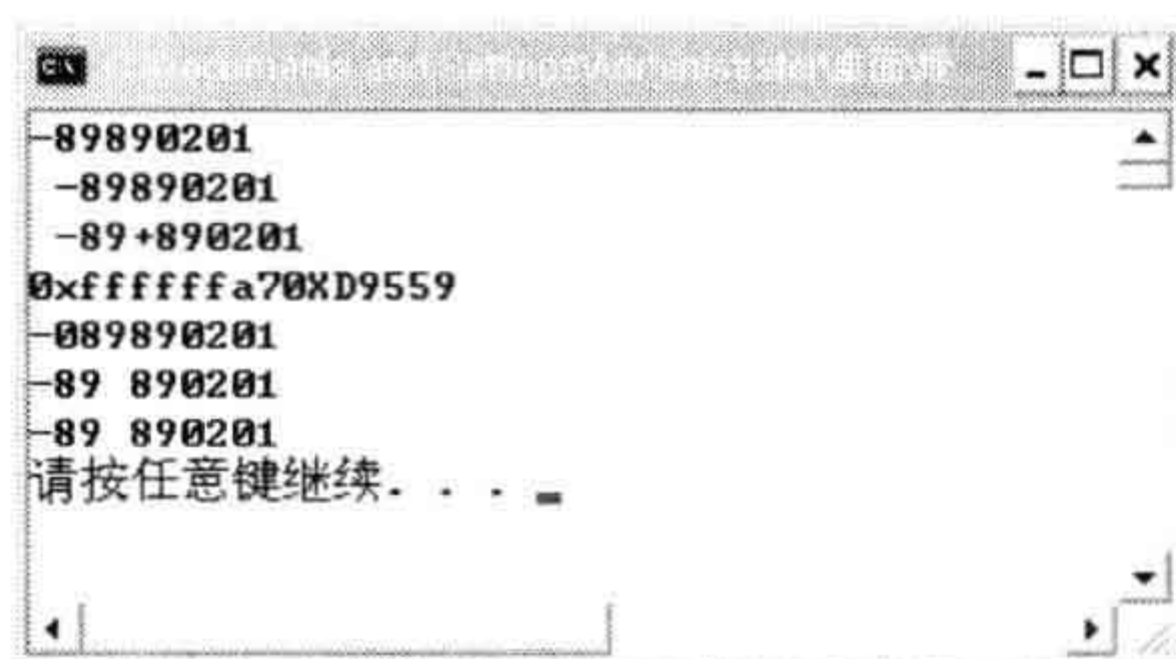


图 6.10 控制宽度的输出结果

由图 6.10 所示的输出结果，就不难理解控制宽度和标志之间的结合问题了。

专家点评

对于宽度的控制，还要根据实际数据的类型宽度而定，最好不要超过你所定义的数据类型的宽度。

问题 75 如何控制输出精度？

问题阐述

对于一些特殊的项目，如金融等，对数据的精度都是有要求的，那么又该如何控制精度呢？

专家解答

说到 C 语言中的精度，就不得不说一下精度修饰符“.”（其后面跟十进制整数）。怎么用？先来看一下示例程序。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
```




```
int a=890201;
float b=8.9020112;
printf("a=%.8d\n",a);
printf("b=%.3f\n",b);
printf("b=%07.2f",b);
system("PAUSE");
return 0;
}
```

上述程序运行结果如图 6.11 所示。

从上述程序及其运行结果不难得出精度修饰符的意义，即如果输出的是整数，则输出最小位数，若输出数的位数小于该值，则添加前置 0；如果输出的是小数，则表示小数的位数；如果输出的是字符，则表示输出字符的个数，若实际位数大于所定义的精度数，则截去超长的部分。

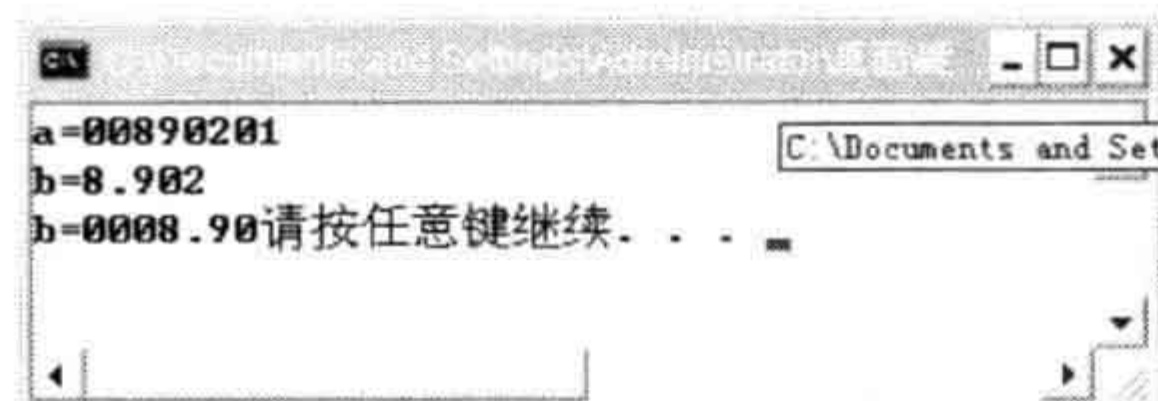


图 6.11 精度控制输出结果

专家点评

对于所有的程序语言来说，在实现的时候都会对精度提出要求。在编写程序时，读者一定一定要注意这个问题。

问题 76 如何控制输出长度？

问题阐述

什么是长度？如何控制长度？

专家解答

其实这里所说的长度表示输出数据占用的内存长度。通俗地说，就是输出指定类型的数据。比如最为常见的%d，表示输出 int 型数据，但是要输出 long 类型的数据，那就要使用%ld表示了。这里将相关的长度修饰符以表格的格式给出，如表 6.3 所示。

表 6.3 长度修饰符

长度修饰符	含 义
h	与整数格式一起使用，表示一个短整型或一个无符号短整型的数据
hh	与整数格式一起使用，表示一个signed char 或unsigned char型的数据
l	与整数格式一起使用，表示一个长整型或一个无符号长整型的数据
ll	与整数格式一起使用，这是长度修饰符l的C99表示方式
L	与实数格式一起使用，表示一个长双精度类型的数据





专家点评



Note

对于%d来说,在32位字长的计算机系统中,int和long这两种数据类型占有相同的字节数,因此%d和%ld两个格式符都可以输出long型的数据,而对于short和char数据类型都可以使用%d格式符进行输出。

问题 77 如何动态设置输出宽度和精度?

问题阐述

前面讲述的都是在编程阶段对输出宽度和精度进行设置,那么如何动态设置输出的精度和宽度呢?

专家解答

其实输出宽度和精度是可以使用“*”号来代替的。这样,printf()函数将会对应地以输出列表中的数据作为输出的宽度和精度。其基本格式如下:

```
printf("%*.*fn",m,n,f);
```

输出列表中的变量m对应的是第一个“*”号,设置的是输出宽度;变量n对应的是第2个“*”号,设置的是输出精度。下面来看一个示例程序。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int m,n;
    float a=89.02011225;
    printf("请输入要输出的最小宽度: \n");
    scanf("%d",&m);
    printf("请输入要输出的小数点位数: \n");
    scanf("%d",&n);
    printf("%*.*fn",m,n,a);
    system("PAUSE");
    return 0;
}
```

程序运行结果如图 6.12 所示。

从程序的运行结果中不难看出,输出宽度为6位,小数点位数设定为2位,加上小数点占1位,总共占5位,所以在输出结果前空出了1位,以保证宽度是6位。

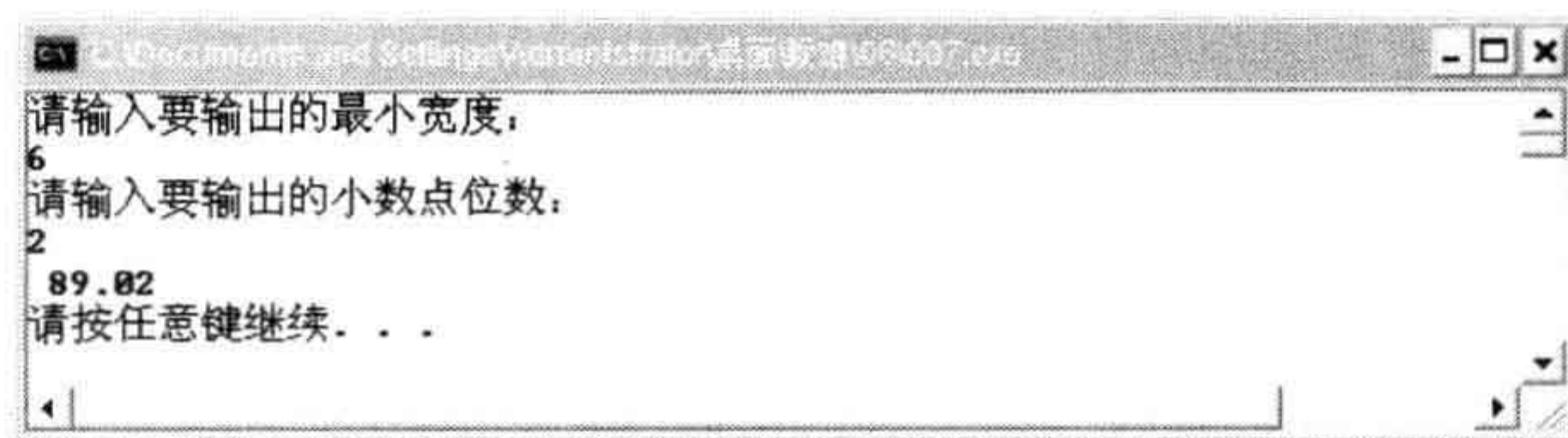


图 6.12 动态控制宽度和精度



Note

专家点评

动态地设置输出数据的宽度和精度，对于开发技能无疑是一种很大的提升。用户可以根据实际需要来设定要输出数据的宽度和精度，更好地控制数据的输出。

问题 78 printf()函数的返回值是什么？

问题阐述

C 语言函数一般都会有一个返回值，返回值是由函数返回给调用程序的一个值，那么 printf() 函数有吗？是什么？

专家解答

printf() 函数也是有返回值的，一般是返回它所输出的数据的字符数目；如果 printf() 函数执行失败，则会返回一个负数。

大多数情况下，程序是不需要处理 printf() 函数的返回值的，一般是在检测 printf() 函数是否执行成功的时候来查看，处理这个返回值。

下面是一个处理 printf() 函数返回值的程序示例，代码如下。

```
#include<stdio.h>
#include<stdlib.h>
#define renum "welcome to mr"
int main()
{
    int a;
    a=printf("%s\n",renum);
    printf("printf 函数的返回值是: %d\n",a);
    system("PAUSE");
    return 0;
}
```

程序运行结果如图 6.13 所示。

上面程序中，首先宏定义了一个 14 位的字符串，然后使用正型变量 a 来接收 printf() 函数的返回值，并将字符输出，同时也输出了返回值，正好是字符串的长度。



Note

专家点评

从这个问题我们不难得出,对于一些看似在应用中不重要的内容,在开发和调试中往往会起到很大的作用,所以读者还是要系统、完善地进行学习。

问题 79 如何理解输出列表?

问题阐述

上面在介绍 printf() 函数基本格式时,后面有个输出列表,如何理解呢?

专家解答

printf() 函数的输出列表可以有多个表达式,数量上至少应该和前面的数据格式符相匹配。如果其数量大于格式符的数量,将不会输出多余的部分;如果数量少于格式符,则会输出一些不可预料的值。

例如:

```
printf("%f,%d\n",f);
```

上面的语句在输出时,只会按照 %f 的格式输出变量 f 和一个不可知的数值。而下面的语句:

```
printf("%f,%d\n",f,i,j);
```

将会输出 %f 格式的变量 f 和 %d 格式的变量 i, 而变量 j 将不会输出。

专家点评

其实,在使用 printf() 函数时,编译器是将会传递给 printf() 函数的数据放到一个堆栈中。堆栈的原则是“后进先出”,也就是说最后保存的数据将先读出来。

问题 80 函数 scanf() 的基本格式是什么?

问题阐述

与格式输出函数 printf() 相对应的是格式输入函数 scanf(), 那么该函数的基本格式是



图 6.13 printf() 函数的返回值



什么?

专家解答

scanf()的功能是按照格式控制的要求, 将从终端输入的数据赋给地址列表中的各个变量。虽然 scanf()的格式控制含义与 printf()相类似, 但是在格式控制中出现的, 除了格式控制符之外的其他字符不能被输出。

scanf()函数的一般格式如下:

scanf(格式控制, 地址列表)

通过 scanf()函数的一般格式可以看出, 参数中的格式控制与 printf()函数相同。例如, %d 表示十进制的整型, %c 表示的是单字符。而地址列表中, 此处应该给出用来接收数据的变量的地址。如得到一个整型数据的操作。

```
scanf("%d",&iInt); /*得到一个整型数据*/
```

地址列表给出各变量的地址, 地址是由取地址运算符“&”后跟变量名组成的。在上面的代码中, &符号是表示取 iInt 变量的地址, 所以变量的地址不用关心具体是多少, 只要像上述代码中那样, 在变量的标识符前加&符号, 就表示取变量的地址。

专家点评

在编写程序时要注意的是, 在函数 scanf()参数的地址列表处, 一定要使用变量的地址, 而不是变量的标识符, 否则编译器会提示错误。

问题 81 scanf()函数的格式字符是什么?

问题阐述

上面探讨了 printf()函数格式字符(以下简称格式符)等问题, 那么与之对应的 scanf()函数的格式符是什么样的呢?

专家解答

首先对 scanf()函数的格式符作一介绍, 具体如表 6.3 所示。

表 6.3 scanf()函数格式字符

格 式 字 符	功 能 说 明
d或,i	用来输入有符号的十进制整数
u	用来输入无符号的十进制整数
o	用来输入无符号的八进制整数
x,X	用来输入无符号的十六进制整数(大小写作用是相同的)



Note



续表

格式字符	功能说明
c	用来输入单个字符
s	用来输入字符串
f	用来输入实型，可以用小数形式或者指数形式输入
e,E,g,G	与f作用相同，e与f、g之间可以相互替换（大小写作用相同）

说明：

格式字符%s 的功能是用来输入字符串。其中将字符串送到一个字符数组中，在输入时以非空白字符开始，以第一个空白字符结束。字符串以串结束标志'\0'作为最后一个字符。

在下面的例子中，将使用格式输入函数 scanf()得到用户输入的两个整型数据（因为 scanf()函数只能用来进行输入操作，所以在屏幕上显示信息时使用显示函数）。

```
#include<stdio.h>
int main()
{
    int iInt1,iInt2;                /*定义两个整型变量*/
    puts("请输入两个整数数值:");    /*通过 puts()函数输出提示信息的字符串*/
    scanf("%d%d",&iInt1,&iInt2);      /*通过 scanf()得到输入的数据*/
    printf("输入的第一个整数为 : %d\n",iInt1);    /*显示第一个输入的数据*/
    printf("输入的第二个整数为 : %d\n",iInt2);    /*显示第二个输入的数据*/
}
```

(1) 为了能接收用户输入的整型数据，在上述代码中定义了两个整型变量 iInt1 和 iInt2。

(2) 因为 scanf()函数只能接收用户输入的数据，而不能显示信息，因此先使用 puts()函数输出一段字符表示提示信息。puts()函数在输出字符串之后会自动进行换行，这样就省去使用换行符。

(3) 调用 scanf()函数。在其参数中可以看到，在格式控制的位置使用双引号将格式字符括住（%d 表示输入的为十进制的整数）；在地址列表位置，使用&符号表示变量的地址。

(4) 此时变量 iInt1 和 iInt2 已经得到了用户输入的数据，调用 printf()函数将变量进行输出。这里要注意的是，printf()函数使用的是变量的标识符，而不是变量的地址；scanf()函数使用的是变量的地址，而不是变量的标识符。

说明：

程序是怎样将输入的内容分别保存到指定的两个变量中的呢？原来，scanf()函数使用空白字符来分隔输入的数据。这些空白字符包括空格、换行、制表符（Tab）。例如，在本程序中，使用换行作为空白字符。

程序运行结果如图 6.14 所示。

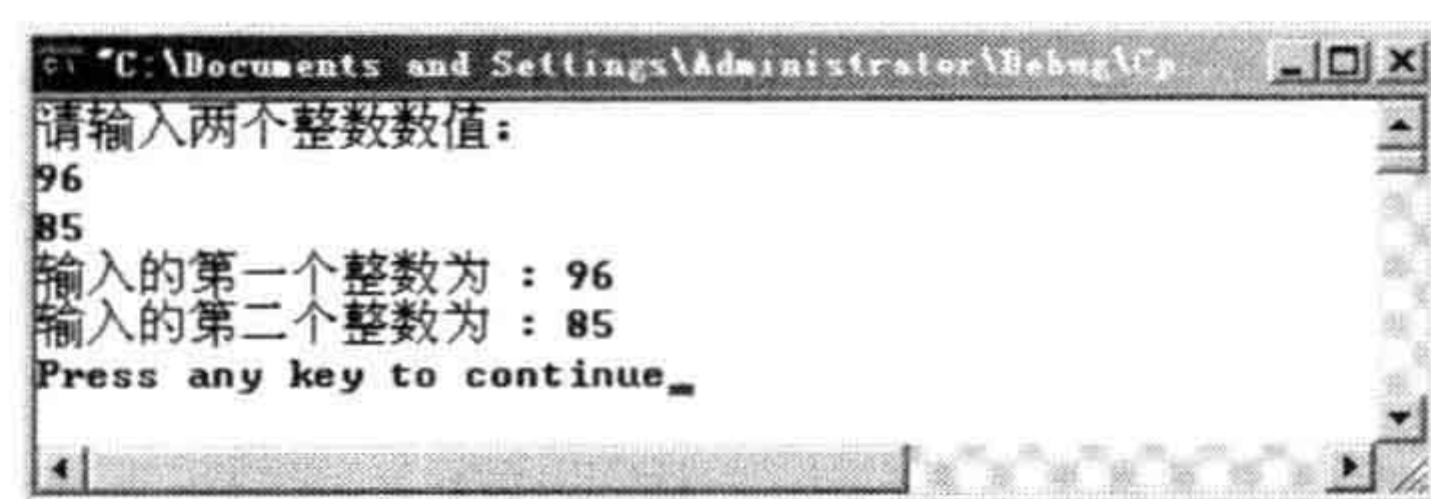


图 6.14 使用格式输入函数 scanf()得到用户输入的数据

在 printf()函数中,除了有格式字符外,还有附加格式用于进行更为具体的说明。相应地, scanf()函数中也有附加格式用于进行更为具体的格式说明,如表 6.4 所示。

表 6.4 scanf()函数的附加格式

字 符	功 能 说 明
l	用于输入长整型数据(可用于%ld, %lo, %lx, %lu)以及double型的数据(%lf或%le)
h	用于输入短整型数据(可用于%hd, %ho, %hx)
n (整数)	指定输入数据所占宽度
*	表示指定的输入项在读入后不赋给相应的变量

下面使用附加格式说明 scanf()函数的格式输入,并对比输入前后的结果,观察其附加格式的效果。

```
#include<stdio.h>
int main()
{
    long iLong;                /*长整型变量*/
    short iShort;              /*短整型变量*/
    int iNumber1=1;            /*整型变量,为其赋值为 1*/
    int iNumber2=2;            /*整型变量,为其赋值为 2*/
    char cChar[10];            /*定义字符数组变量*/
    printf("请输入一个长整型变量数值\n"); /*输出提示信息*/
    scanf("%ld",&iLong);        /*输入长整型数据*/
    printf("请输入一个短整型数值\n");    /*输出提示信息*/
    scanf("%hd",&iShort);      /*输入短整型数据*/
    printf("请输入一个整数:\n");        /*输出提示信息*/
    scanf("%d*d",&iNumber1,&iNumber2); /*输入整型数据*/
    printf("请输入一个字符串,但是输出时只能显示前三个字符\n"); /*输出提示信息*/
    scanf("%3s",cChar);            /*输入字符串*/
    printf("长整型的数值为: %ld\n",iLong); /*显示长整型值*/
    printf("短整型的数值为: %hd\n",iShort); /*显示短整型值*/
    printf("整型数值 1 为: %d\n",iNumber1); /*显示整型 iNumber1 的值*/
    printf("整型数值 2 为: %d\n",iNumber2); /*显示整型 iNumber2 的值*/
    printf("输出字符串的前三位: %s\n",cChar); /*显示字符串*/
}
```

(1) 为了使程序中的 scanf()函数能接收数据,要在程序代码中定义所使用的变量。为了演示不同格式说明的情况,定义变量的类型有长整型、短整型和字符数组。





(2) 使用 `printf()` 函数显示一串字符, 提示输入的数据为长整型, 调用 `scanf()` 函数使变量 `iLong` 得到用户输入的数据。在 `scanf()` 函数的格式控制部分, 使用附加格式字符 `l` 表示长整型。

(3) 再使用 `printf()` 函数显示提示信息, 提示输入的数据为短整型。调用 `scanf()` 函数时, 使用附加格式字符 `h` 表示短整型。

(4) 使用格式字符 `*` 的作用是表示指定的输入项在读入后不赋给相应的变量。在代码中分析这句话的含义就是: 第一个 `%d` 是输入 `iNumber1` 变量, 第二个 `%d` 是输入 `iNumber2` 变量, 但是在第二个 `%d` 前有一个 `*` 附加格式说明字符, 这样第二个输入的值被忽略, 也就是说 `iNumber2` 变量不保存相应输入的值。

(5) `%s` 是用来表示字符串的格式字符, 将一个数 `n` (整数) 放入 `%s` 中间, 这样就指定了数据的宽度。在程序中, `scanf()` 函数中指定的数据宽度为 3, 那么在输入一个字符串时, 只是接收前 3 个字符。

(6) 最后利用 `printf()` 函数将输入得到的数据进行输出。

程序运行结果如图 6.15 所示。

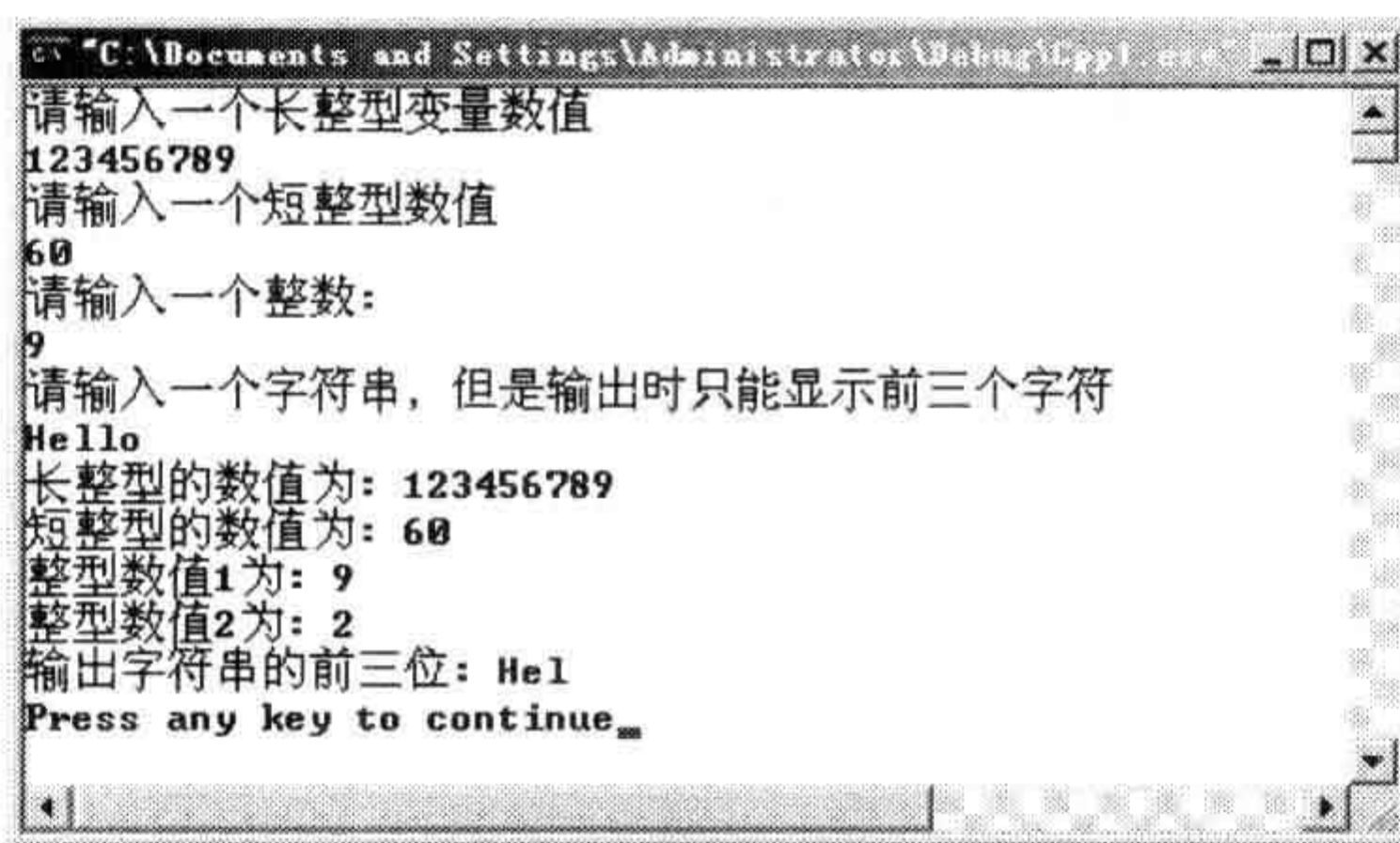


图 6.15 使用附加格式进行说明 `scanf()` 函数的格式输入

专家点评

一个 `scanf()` 函数也可以输入多个数据, 多个连续的 `scanf()` 函数同时输入数据的时候和一个 `scanf()` 函数的调用效果相同, 在使用时最好在每个输入格式符之间加上空格区分。

问题 82 使用 `scanf()` 函数应注意的问题是什么?

问题阐述

上面介绍了 `scanf()` 函数的基本格式和格式符, 那么在使用时应注意什么问题呢?

专家解答

那么具体该注意哪些问题呢? 下面来看一下比较常见的一些错误。

1. 不能控制精度

与 `printf()` 函数不同, `scanf()` 函数输入实数时, 是不能控制精度的。例如, 下面的函数



语句就是不正确的。

```
scanf("%5.2f",&a);
```

2. 在格式字符串中包含非格式字符

前面介绍了在输入多个数值数据时,若格式字符串中没有非格式字符作为数据之间的间隔,则可以使用空格作为间隔。scanf()函数在遇到空白字符或者是非法字符时就会认为当前数据结束。

如果遇到非格式字符,一定要原样输入,这一点必须注意。

专家点评

以上两点是最常犯的错误,希望读者在使用时多加注意。

问题 83 scanf()函数的返回值是什么?

问题阐述

scanf()函数是用于数据输入的,输入变量的值被改变,那么 scanf()函数本身是否有返回值?返回值有什么意义呢?

专家解答

scanf()函数的返回值很少有人用到,它是一个整数,用于表示成功输入数据的个数。来看下面的例子。

```
#include<stdio.h>
main()
{
    int a,b,c;
    printf("输入三个整数: ");
    scanf("%d%d%d",&a,&b,&c);
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

程序运行结果如图 6.16 所示。

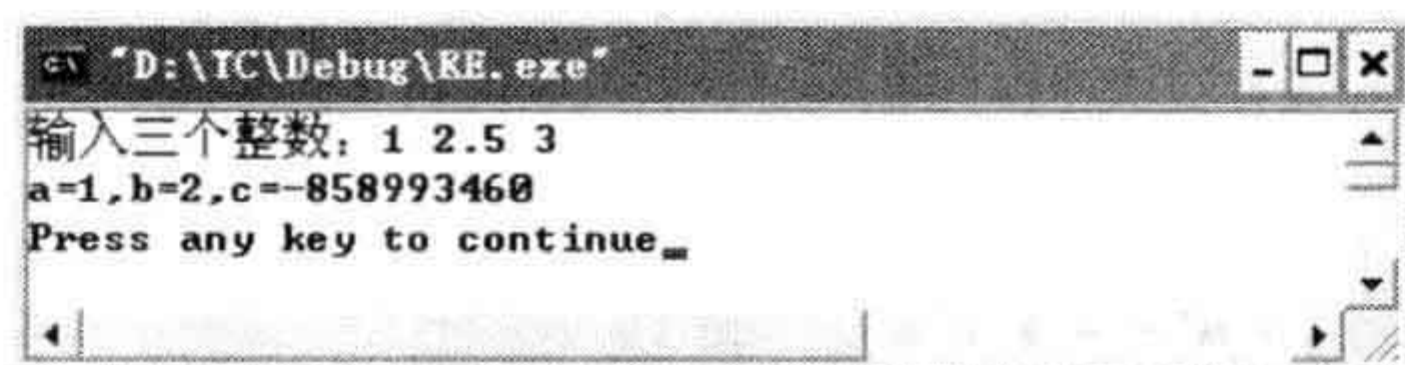


图 6.16 scanf()函数的错误输入

程序中,由于对整数 b 试图输入 2.5 而导致错误。实际上, b 只能得 2,“.5”后的所有数据输入出错,不能正确赋值,因此 c 得不到正确结果。如果程序在这个基础上继续运



Note



行, 会导致后面的错误越叠加越多。怎样由程序本身检查出这种错误呢? 这时就可以用 `scanf()` 函数的返回值来实现。

修改一下程序, 取出 `scanf()` 函数返回值。如果它的值是 3, 那么表示所有数据均已正确输入; 如果小于 3, 必定出现错误输入。

上面程序可修改为:

```
#include<stdio.h>
main()
{
    int a,b,c,n=0;
    printf("输入三个整数: ");
    while(n!=3)
    {
        n=scanf("%d%d%d",&a,&b,&c);
        if(n<3)
            printf("数据格式错误, 请重新输入.\n");
        fflush();
    }
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

程序运行结果如图 6.17 所示。

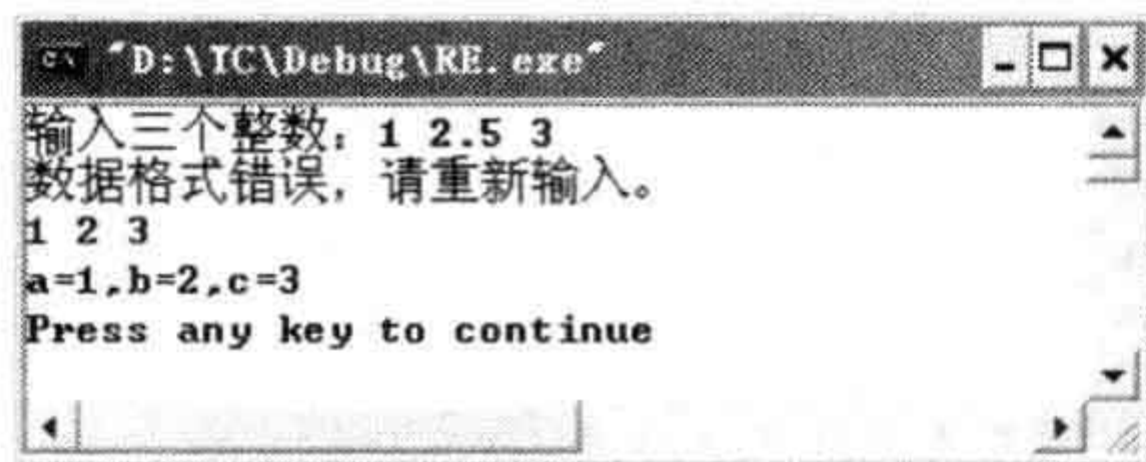


图 6.17 加入错误检查的 `scanf()` 函数输入

专家点评

输入数据格式错误问题, 很多初学者都会忽略, 认为输入时总是没错的。通过 `scanf()` 函数的返回值, 可以很好地解决这一问题。

问题 84 如何使用 `getchar()` 函数?

问题阐述

字符数据输入使用的是 `getchar()` 函数, 那么该如何使用该函数呢?

专家解答

`getchar()` 函数的作用是从终端 (输入设备) 输入一个字符。该函数与 `putchar()` 函数不





同之处是它没有参数。

该函数的定义如下。

```
int getchar();
```

使用 `getchar()` 函数时也要添加头文件 `stdio.h`，函数的值就是从输入设备得到的字符。例如，从输入设备得到一个字符赋给字符变量 `cChar`。

```
cChar=getchar();
```

`getchar()` 函数可作为 `putchar()` 函数的参数，当 `getchar()` 函数从输入设备得到字符，后由 `putchar()` 函数将字符输出。

如要实现字符数据的输入，可使用 `getchar()` 函数获取在键盘上输入的字符，再利用 `putchar()` 函数进行输出。下例演示了将 `getchar()` 函数作为 `putchar()` 函数表达式的一部分，进行输入和输出字符的方式。

```
#include<stdio.h>
int main()
{
    char cChar1;                /*声明变量*/
    cChar1=getchar();           /*在输入设备得到字符*/
    putchar(cChar1);            /*输出字符*/
    putchar('\n');              /*输出转义字符换行*/
    getchar();                  /*得到回车字符*/
    putchar(getchar());         /*得到输入字符，直接输出*/
    putchar('\n');              /*换行*/
    return 0;                  /*程序结束*/
}
```

(1) 要使用 `getchar()` 函数，首先要包括头文件 `stdio.h`。

(2) 声明变量 `cChar1`，通过 `getchar()` 函数得到输入的字符，赋值给 `cChar1` 字符型变量。然后使用 `putchar()` 函数将变量进行输出。

(3) 使用 `getchar()` 函数得到输入过程中的 Enter 键。

(4) 在 `putchar()` 函数的参数位置调用 `getchar()` 函数得到字符并输出。

程序运行结果如图 6.18 所示。

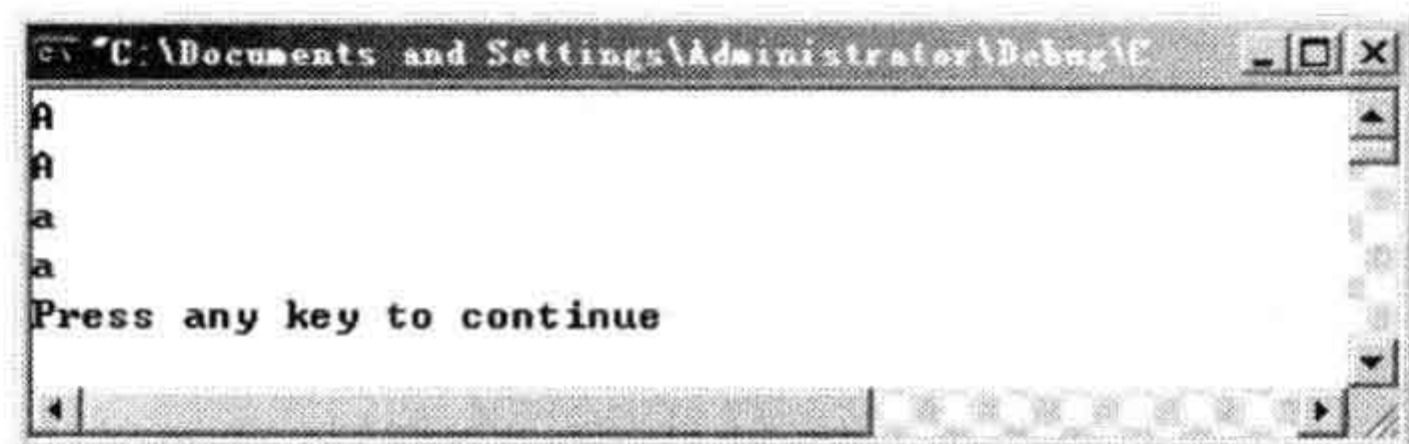


图 6.18 使用 `getchar()` 函数实现字符数据输入

专家点评

需要注意的是，`getchar()` 函数只能接收一个字符。`getchar()` 函数得到的字符可以赋给一



Note



个字符变量或整型变量，也可以不赋给任何变量，还作为表达式的一部分。例如：

```
putchar(getchar());
```



Note

问题 85 getch()函数如何使用？

问题阐述

上面介绍了 getchar()函数，那么 getch()函数又是干什么用的？怎么使用？

专家解答

getch()函数与 getchar()函数的功能基本相同，都是接收用户输入的一个字符。其函数原型如下。

```
int getch(void);
```

getch()函数可以直接从键盘获取输入，无须等待用户按下 Enter 键，只要用户按下一个按键，它立即就会返回。

getch()函数常用于程序的调试阶段，可暂停程序。在此要注意的是，其原型不是在 stdio.h 头文件中，而是在 conio.h 头文件中。

专家点评

getch()函数也有返回值，其返回值是输入字符的 ASCII 码；如果出错，则返回-1。

问题 86 如何应用 gets()函数？

问题阐述

输入字符串使用的是 gets()函数，其作用是将读取的字符串保存在形式参数 str 变量中。那么该如何使用该函数呢？

专家解答

gets()函数将一直读取字符串，直到出现新的一行为止。其中，新的一行的换行字符将会转化为字符串中的空终止符'\0'。gets()函数的定义如下。

```
char *gets( char *str );
```

在使用 gets()函数前，要为程序加入头文件 stdio.h。其中，str 字符指针变量为形式参数。例如，定义字符数组变量 cString，然后使用 gets()函数获取输入字符的方式如下。

```
gets(cString);
```




在上面的代码中, cString 变量获取到了字符串, 并将最后的换行符转化成了终止字符。使用字符串输入函数 gets() 获取输入信息的程序如下。

```
#include<stdio.h>
int main()
{
    char cString[30];           /*定义一个字符数组变量*/
    gets(cString);             /*获取字符串*/
    puts(cString);             /*输出字符串*/
    return 0;                  /*程序结束*/
}
```



Note

(1) 因为要接收输入的字符串, 所以要定义一个可以接收字符串的变量。在上述代码中, 定义 cString 为字符数组变量的标识符。有关字符数组的内容将在后面的章节介绍, 在此只需知道此变量可以接收字符串即可。

(2) 调用 gets() 函数, 其中参数为定义的 cString 变量。调用该函数时, 程序会等待用户输入字符; 当用户输入完字符并按 Enter 键确定时, gets() 函数获取字符结束。

(3) 使用字符串输出函数 puts() 将获取后的字符串进行输出。

程序运行结果如图 6.19 所示。

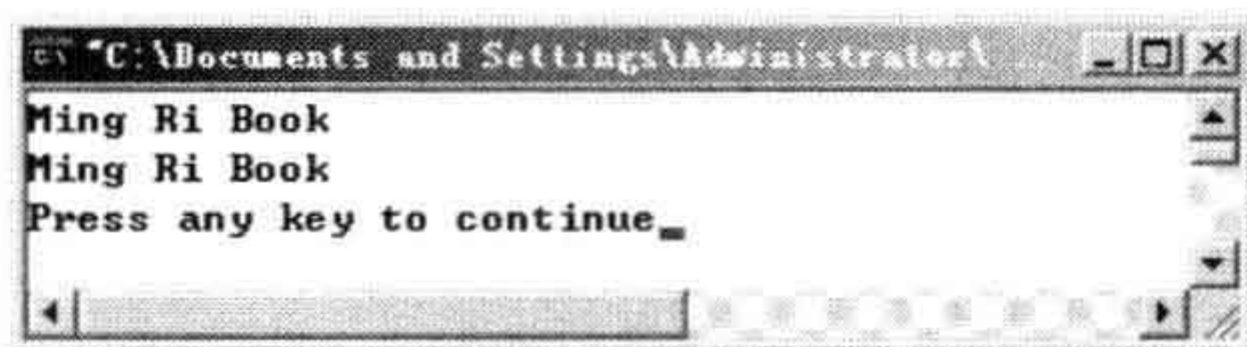


图 6.19 使用字符串输入函数 gets() 获取输入信息

专家点评

从上面可以看出, 当输入的字符串中含有空格时, 输出仍为全部字符串。这说明 gets() 函数并不以空格作为字符串输入结束的标志, 而只以回车作为输入结束。这是与 scanf() 函数不同的。

问题 87 如何应用 putch() 函数?

问题阐述

putch() 函数用于向屏幕中输出字符, 那么该如何使用这个函数呢?

专家解答

putch() 函数也是在头文件 conio.h 中, 其具体语法格式如下。

```
int putch();
```

例如, 输出一个字符 m 的语句就是:



Note

```
putch('m');
```

也可以输出字符变量所保存的字符。例如：

```
putch(c);
```

更重要的是，putch()函数还可以输出控制字符，如换行符。例如：

```
putch('\n');
```

对于控制字符，putch()函数只是执行控制功能，不在屏幕上显示。

专家点评

前面曾经讲过另外一个字符输出函数 putchar(), 它也可以用来向屏幕上输出一个字符。在使用上，该函数与 putch()函数没什么区别。

问题 88 puts()函数该如何应用?

问题阐述

字符串输出使用的是 puts()函数，其作用是输出一个字符串到屏幕上。那么在开发中要如何使用呢？

专家解答

首先来看该函数的定义：

```
int puts( char *str );
```

使用该函数时，先要在程序中添加 stdio.h 头文件。其中形式参数 str 是字符指针类型，可以用来接收要输出的字符串。例如，使用 puts 函数输出一串字符。

```
puts("I LOVE CHINA!"); /*输出一个字符串常量*/
```

这行语句是输出一段字符串，之后会自动地进行换行操作。这与 printf()函数有所不同，在前面的实例中，使用 printf()函数进行换行时，要在其中添加转义字符'\n'进行换行操作。puts()函数会在字符串中判断'\0'结束符，遇到结束符时，后面的字符不再输出并且自动换行。例如：

```
puts("I LOVE\0 CHINA!"); /*输出一个字符串常量*/
```

将上面的语句加上'\0'字符后，puts()函数输出的字符串就变成：I LOVE。

下面使用字符串输出函数显示提示信息，即使用 puts()函数对字符串常量和字符串变量进行操作，从中学习使用 puts()的方式。

```
#include<stdio.h>
int main()
```




```

{
    char* Char="I LOVE CHINA";           /*定义字符串指针变量*/
    puts("I LOVE CHINA!");               /*输出字符串常量*/
    puts("\0 LOVE\0 CHINA!");            /*输出字符串常量, 其中加入结束符'\0'*/
    puts(Char);                           /*输出字符串变量的值*/
    Char="I LOVE\0 CHINA!";              /*改变字符串变量的值*/
    puts(Char);                           /*输出字符串变量的值*/
    return 0;                             /*程序结束*/
}

```



Note

(1) 从上述代码中可以看到, 字符串常量赋值给字符串指针变量。有关字符串指针的内容将会在后面的章节进行介绍。此时可以将其看作整型变量, 为其赋值后, 就可以使用该变量。

(2) 第一次使用 puts() 函数输出字符串常量时, 在该字符串中没有结束符 '\0', 所以输出的字符会一直持续到最后编译器为其字符串添加结束符 '\0' 为止。

(3) 第二次使用 puts() 函数输出的字符串常量中, 为其添加了两个 '\0'。输出的显示结果表明: 检测字符时, 程序遇到第一个结束符便会停止输出字符, 并且进行换行操作。

(4) 第三次使用 puts() 函数输出的是字符串指针变量, 函数根据变量的值进行输出。因为变量的值中并没有结束符, 所以会一直将字符输出到最后编译器为其添加的结束字符, 然后进行换行操作。

(5) 改变变量的值, 再使用 puts() 函数输出变量时, 可以看到由于变量的值中有结束符 '\0', 所以显示结果到第一个结束符后停止, 最后进行换行操作。

程序运行结果如图 6.20 所示。

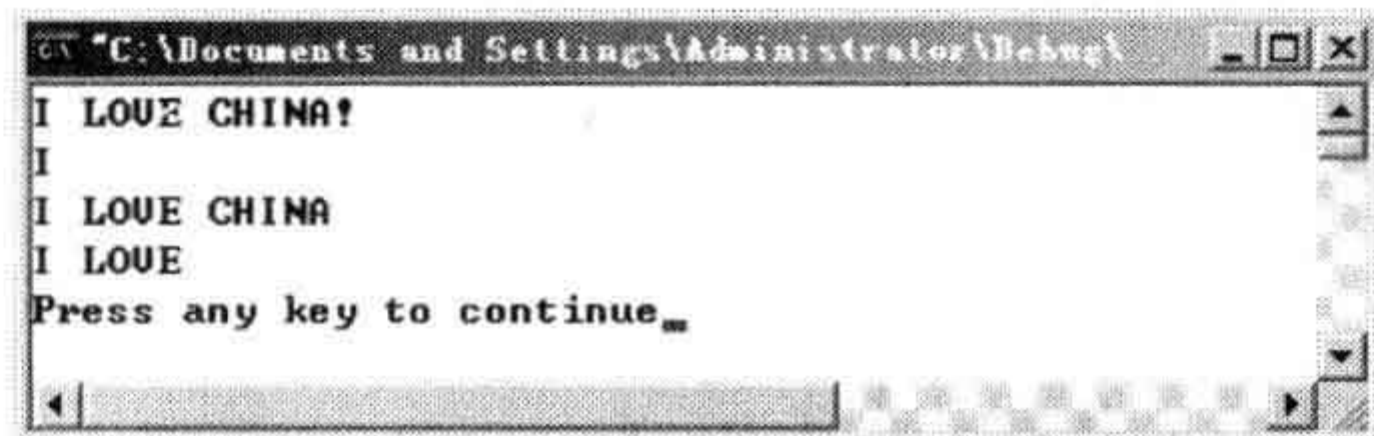


图 6.20 利用字符串输出函数显示提示信息

专家点评

在前面的章节中, 曾经介绍编译器会在字符串常量的末尾添加结束符 '\0', 这也就说明了为什么 puts() 函数会在输出字符串常量时最后进行换行操作。

问题 89 如何控制多数值的输入?

问题阐述

在程序开发中, 有时需要设计一个一次输入多个数值的模块, 那么如何控制呢?



专家解答



Note

`scanf()`函数一次可以输入一个数值，也可以输入多个数值，而且多个数值类型可以相同，也可以不相同。用 `scanf()`输入的多个数据之间用 C 语言标准分隔符分开。标准分隔符包括以下 3 个：空格（下面用□表示）、Enter 键（下面用↵表示）、Tab 键。例如：

```
scanf("%d%d",&a,&b);
```

为 a,b 输入 10 和 20 时，可以有以下几种输入方法。

(1) 10□20

(2) 10↵

20

(3) 10Tab 20

标准分隔符的个数可以是多个，也可以混用。例如，上面空格可以是 3 个，也可以是 5 个，还可以输入几个空格后再输入几个回车。

这是在 `scanf()`的双引号中只有格式说明符的情况，如果还有其他字符呢？输入方式是 `scanf()`语句中有什么，执行时就输入什么，或称“格式声明符之外的其他字符输入时直接输入”。例如：

```
scanf("%d,%d");
```

此时就输入 10,20。

如果输入数据中包含字符型，那么字符型输入时不用分隔符。例如：

```
int a,b;  
scanf("%d%c%d",&a,&c,&b);
```

输入 10x20 结果是：

a 的值是 10，b 的值是 20，c 的值是字母 x。

输入 10□20 结果是：

结果 c 的值就是空格。

专家点评

以上规则可以保证输入多个数的控制。

问题 90 如何输入字符数组？

问题阐述

在程序中，`scanf()`函数可以输入任意类型的数据，`gets()`函数只能输入字符串等，但是如何更好地输入字符数组呢？



专家解答

前面介绍了如何使用格式输入函数 `scanf()`，那么可以使用 `%c` 格式符逐个输入字符。这样输入有些麻烦，但是可以更好地保证输入的准确性。此外，也可以使用 `%s` 格式符整个输入字符串。例如：

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    char str[20];
    printf("请输入一个字符串（20 以内）： \n");
    scanf("%s",str);
    printf("您输入的字符串是： %s\n",str);
    system("PAUSE");
    return 0;
}
```

上述程序运行结果如图 6.21 所示。

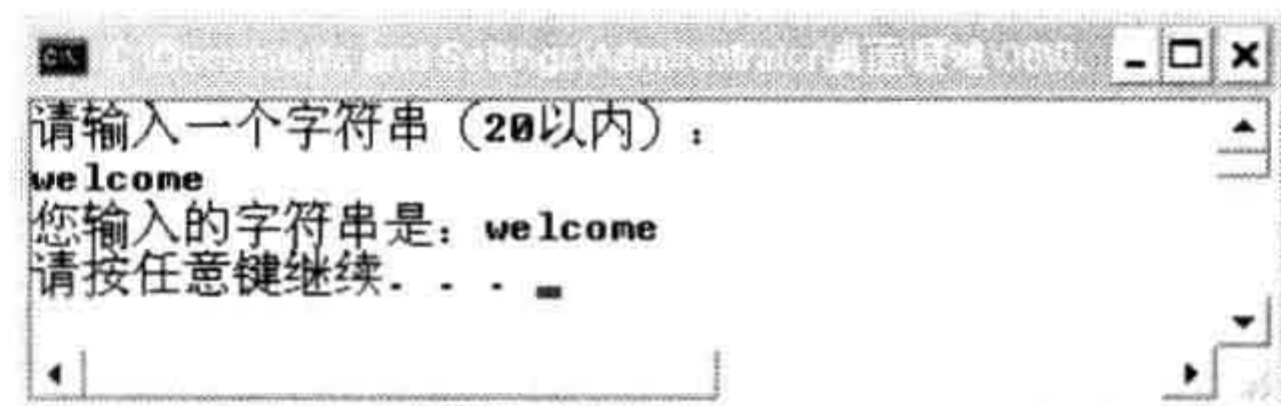


图 6.21 字符串输入

上述程序在整个输入一个字符串以后，按下 `Enter` 键就会将输入的字符串输出到屏幕。在此要注意的是，不要在中间输入空格，否则输出会以空格断开。

专家点评

对字符串的输入方式，读者可以根据自己或用户的需要进行设置，准确性要求高的可以使用循环进行单个输入，否则就进行整个字符串的输入。



Note

第7章

选择、分支结构程序设计

- ▶▶ $5 > 4 > 3$ 为什么不成立——谈谈关系表达式的值
- ▶▶ $=$ 和 $==$ 如何区分?
- ▶▶ 什么叫逻辑短路?
- ▶▶ if 语句的基本形式有哪些? 如何应用?
- ▶▶ 浮点数的相等比较是否可以用 $==$?
- ▶▶ 关系运算符和数学不等号有什么区别?
- ▶▶ if 语句后面一定不能写分号吗?
- ▶▶ 这个程序为什么多执行了好多语句?
- ▶▶ 不用关系表达式和逻辑表达式做条件
- ▶▶ 怎样理解复合语句中的变量?
- ▶▶ 如何进行 if 语句的嵌套?
- ▶▶ 条件运算符“?:”怎样应用?
- ▶▶ switch 语句的基本格式是什么?
- ▶▶ if 语句与 switch 语句的优缺点是什么?
- ▶▶ switch 语句中的 default 关键字是否必须?
- ▶▶ break 关键字在 switch 语句中应注意什么?
- ▶▶ 如何正确判断 if 和 else 的匹配?
- ▶▶ switch 和 case 后的表达式值的类型是否可以浮点型?
- ▶▶ 区段划分型条件有什么技巧?
- ▶▶ default 必须写在所有 case 之后吗?



问题 91 $5>4>3$ 为什么不成立——谈谈关系表达式的值

问题阐述

$5>4>3$ 不成立，这怎么可能呢？

专家解答

请您记好了，这个结论在数学中成立，但是在 C 语言中却不成立。

这是一个 C 语言关系表达式的值的问题。

对表达式 $5>4>3$ 的值，先给出四个选项：

A. true B. false C. 1 D. 0

答案应该是 D。请看下面的分析：

C 语言没有专门逻辑值 true、false，一个关系表达式或逻辑表达式的执行结果成立时用 1 表示，不成立时用 0 表示。但不是只有 0 和 1 可以表示逻辑值，任何数都可以。其他别的值表示逻辑值时，除了 0 之外的所有数都是真，整数可以，小数也可以，甚至字符也可以，把 100、3.5、'a' 当成逻辑值看待时，都是“真”。

关系表达式 $5>4>3$ ，应怎样理解？这里还有一个运算符结合性的问题，C 语言的运算符有两个特征，即优先级和结合性。

可以通过下面两个例子来理解优先级和结合性的概念。

$3+2>6$

$a>=0 \ \&\& \ a<=100$

C 语言中规定：所有算术运算符优先级高于所有关系运算符，所有关系运算符高于逻辑运算符“&&”和“||”。

因此以上两个表达式的正确执行顺序是 $(3+2)>6$ ， $(a>=0) \ \&\& \ (a<=100)$ 。

结合性只应用在优先级相同的情况下。除赋值(=)之外的所有双目运算符都是从左向右结合的，即 $100/10/2$ 应理解为 $(100/10)/2$ ，而不理解为 $100/(10/2)$ 。

现在来解答最初的问题：

$5>4>3$

它应该理解为 $(5>4)>3$

$5>4$ 成立，值为 1，那下一步就是 $1>3$ ，结果不成立。

再看以下两个表示式的值：

I. $3+2>5$ II. $3+(2>5)$

结论：I 是逻辑值假，值为 0，II 是一个数，值为 3。

请读者朋友自己分析一下原因。

再看：!! a 一定等于 a 吗？

结论：不一定，只有 a 等于 0 或等于 1 时成立，其他值均不成立。



Note



专家点评

上述所有问题的根源都在于 C 语言用数字表示逻辑值，即非 0 表示假，0 表示真。

知道这些细节，对你的程序设计是很有用的，如果你想编程测试一个变量 a 的值在 $0 \sim 100$ 之间，一定不要写 $0 \leq a \leq 100$ ，这样对 a 是负数的取值都会认为是成立的。正确的写法是 $0 \leq a \ \&\& \ a \leq 100$ 。



Note

问题 92 $=$ 和 $==$ 如何区分？

问题阐述

这两个符号，在不严格的情况下，都可以读成“等于”，但是两个“等于”在程序中有完全不同的用法。错误使用将导致程序无法得出正确的结果，而且有时没有错误提示。那么，怎样正确区分这两个运算符呢？

专家解答

这两个符号一个是赋值运算符“ $=$ ”，另一个是关系运算符或称比较运算符“ $==$ ”，表示相等。有些老师讲课时会特别强调读法，“ $a=b$ ”读作“将 b 赋值于 a ”，“ $a==b$ ”读作“ a 等于 b ”，但大多数学生自己读的时候都是统统读成“ a 等于 b ”，只有发言时才强制自己读“赋值”的。

关系运算符“ $==$ ”与习惯思维的等于相同。以“ $a==b$ ”为例，是看 a 与 b 是否相等，要得到结论，成立或不成立，应用在条件中“如果 a 与 b 相等，去做某一操作”对应程序就是 `if(a==b).....`

而“ $=$ ”是个赋值运算符。“ $a=b$ ”是“将 b 的值赋于 a ”，或称“让 a 的值等于 b 的值”。注意，此时 a 的被改变，将造成一个即成事实。而“ $a==b$ ”是看 a 与 b 是否相等， a 的值不被改变。在程序中用的最多的就是“ $=$ ”。

下面分析两个错误的用法。

```
int a,b;
a=3;
b=4;
if(a=b)
    printf("相等");
else
    printf("不等");
```

上面程序显示的是“相等”还是“不等”呢？不要不假思索地回答“不等”，应该是“相等”。上面程序可理解为：如果条件“ $a=b$ ”成立，就显示“相等”，不成立就显示“不相等”。那这个条件怎么可能是成立的呢。“将 b 的值赋于 a ”，注意不是“ a 是否等于 b ”。



“将 b 的值赋于 a” 算是什么条件呢, 这就要理解 C 语言中什么是“真”, 什么是“假”, 赋值表达式的值这两个概念。

C 语言规定:

(1) C 语言中没有专门的逻辑值, 任何数据都可以表示逻辑值, 规定 0 是假, 除 0 外的任何其他数都是真。

(2) 赋值表达式的值就是变量的值。

因此, 赋值表达式 $a=b$ 的值是 3, 再将 3 用在条件中, 当成逻辑值去理解, 结果是真。因此, 上面程序的执行结果显示“相等”。

这样的程序在编译时系统会给出一个警告信息。如果不理会警告, 再编译一次, 就成功通过了。

专家点评

区分=和==的应用领域, 这是一个基本问题, 用的多了, 自然也就懂了。但要理解一个莫名其妙的程序的执行结果, 有时用到很多 C 语言的细节知识。

问题 93 什么叫逻辑短路?

问题阐述

只有电路中才听说过短路。逻辑短路, 不像是个好词儿。这个词不是描述人的, 是描述 C 语言程序的。它是程序执行时的一种特殊的跳跃状态。那么逻辑短路具体是什么呢?

专家解答

C 语言中, 表示条件时用关系表达式, 即由关系运算符 $>$ 、 $>=$ 、 $<$ 、 $<=$ 、 $==$ 、 $!=$ 六个运算符之一连接起来的表达式。当表示复杂条件时, 可以将多个关系表达式用逻辑运算符连接, 构成逻辑表达式。

C 语言中的逻辑运算符有 3 个:

$\&\&$ 用于连接两个条件, 当这两个条件同时成立时, 整个条件成立。

$\|$ 用于连接两个条件, 这两个条件只要有一个成立, 整个条件就成立。

! 只在其后连接一个表达式, 取与这个表达式结果相对的逻辑值, 即真变假, 假变真。

在 $\&\&$ 的使用中, 如: $a>0\&\&a\leq 100$, a 的值如果等于 -5, 这时条件的前一部分就不成立, 对于 $\&\&$ 来说, 不管后面的条件是否成立, 整个条件都不会成立 (这两个条件同时成立时, 整个条件就成立)。或者说, $\&\&$ 后面的条件执不执行都不会改变整个表达式的取值, 这样后半部分实际也就没必要执行了。在 C 语言中, 既然没必要执行, 那就不执行。这种一部分代码被跳过、不执行的情况就叫逻辑短路。

$\&\&$ 可以构成逻辑短路, $\|$ 也可以, $\|$ 的功能是两个条件只要有一个成立, 整个条件就成立。那如果前半条件就成立, 后半也不用执行了, 这就是由 $\|$ 构成的逻辑短路。



Note



只有这两种形式的逻辑短路。下面看一段程序。

```
#include<stdio.h>
main()
{
    int a=1,b=2,c=3,d=4,e=5;
    if(a>b&& c++>d)
        e++;
    printf("%d, %d",c,e);
}
```

运行结果如图 7.1 所示。

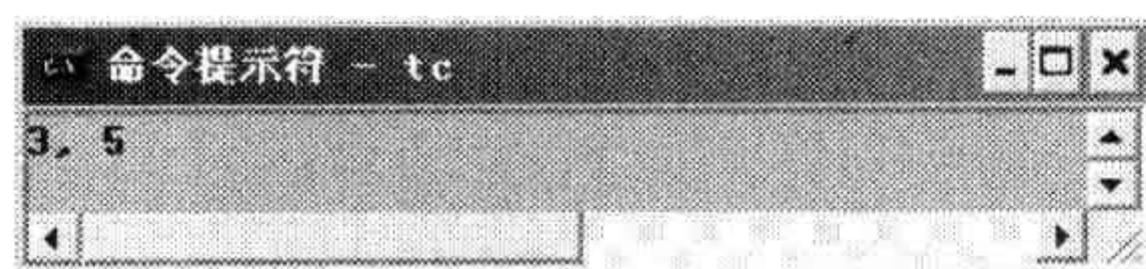


图 7.1 逻辑短路运行结果

因为 $a > b$ 不成立，后面的 $c++ > d$ 没获得执行，因此 c 仍为 3，而 if 条件不成立， $e++$ 没执行， e 的值仍为 5。

如果程序中 if 行改为 $\text{if}(a < b \&\& c++ > d)$ ，其他不变，结果应为“4, 5”。因为此时 $a < b$ 成立，对 $\&\&$ 来说，前半部分成立，不能决定整个表达式的值（如果后半部分成立，整体就成立，后半部分不成立，整体就不成立），不构成逻辑短路。因此后半部分还要执行，结果为 4, 5。

专家点评

在计算平均分是否及格的程序中，前面程序经运算得出 n 为科目数， s 为总成绩， n 有可能得 0。程序可以写成：

```
if(n>0&& s/n>60)
...
```

这样的程序在 C 语言中是正确的，在其他没有逻辑短路的语言中（如 VB）是错误的。如果没有逻辑短路，以上程序就会存在安全隐患，因为当 $n=0$ 时，就会出现除 0 错，系统直接异常退出。有逻辑短路做保障，以上程序可以在 C 语言中正确运行。

问题 94 if 语句的基本形式有哪些？如何应用？

问题阐述

各种编程语言都有 if 语句，C 语言中的 if 语句有哪些基本形式？怎样应用这些基本形式？



专家解答

细致来说, if 语句的形式有 3 种:

(1) 单分支结构。

```
if(条件)
{语句}
```

用于“条件”成立时执行“语句”, 不成立时不执行任何操作。

例: 如果变量 a 是小写字母, 就把它变成大写, 这样不论大写还是小写, 就都变成大写了。程序如下:

```
#include<stdio.h>
main()
{
    char a;
    clrscr();
    printf("input a character:");
    scanf("%c",&a);
    if(a=='a'&&a<='z')
        a=a-32;
    printf("%c",a);
}
```

以上程序可以实现输入的字母不论是 大写还是小写, 都显示成大写字母的功能。

(2) 双分支结构。

```
if(条件)
{语句 1}
else
{语句 2}
```

用于特定条件成立时执行第一种方案, 不成立时执行第二种方案。

例: 根据输入的商品价格、数量求金额, 如果数量超过 20, 价格打八折, 不超过 20 不打折。程序如下。

```
#include<stdio.h>
main()
{
    float a,b,c;
    printf("商品价格:");
    scanf("%f",&a);
    printf("购买数量:");
    scanf("%f",&b);
    if(b>=20)
        c=a*b*0.8;
    else
        c=a*b;
```



Note



```
printf("应付金额%.2f 元",c);  
}
```

(3) 多分支结构。

```
if(条件)  
{语句 1}  
else if(条件 2)  
{语句 2}  
...  
else if(条件 n)  
{语句 n}  
else  
{语句 n+1}
```

用于多个不相融的条件。

例：根据 0~100 的考试成绩，分成优秀、良好、及格、不及格四个级别。程序如下：

```
#include<stdio.h>  
main()  
{  
    float x;  
    printf("输入考试成绩");  
    scanf("%f",&x);  
    if(x<0||x>100)  
        printf("请输入 0-100 之间的数");  
    else if(x>=85)  
        printf("优秀");  
    else if(x>=70)  
        printf("良好");  
    else if(x>=60)  
        printf("及格");  
    else  
        printf("不及格");  
}
```

专家点评

这是一个基本问题。只有掌握基本语句写法，才能写出更复杂的综合项目来。

问题 95 浮点数的相等比较是否可以用==?

问题阐述

“==”用于比较两个表达式是否相等。在程序中，是否可以这样写“if(a==2.8)”（a 是一个已定义的 float 型变量）？





专家解答

初看起来没什么不可以，但是这种写法是绝对不可以的。原因如下：

计算机中的数都是用二进制表示的，如十进制数 3.625，可表示为二进制数 11.101。
转换过程如下：

(1) 整数部分。

$$3 \div 2 = 1 \dots\dots 1$$

$$1 \div 2 = 0 \dots\dots 1$$

最后的二进制数是所有余数从后向前写。 $(3)_{10} = (11)_2$

(2) 小数部分。

$$0.625 \times 2 = 0.25 + 1$$

$$0.25 \times 2 = 0.5 + 0$$

$$0.5 \times 2 = 0 + 1$$

最后的二进制数是所有整数从前向后写。 $(0.625)_{10} = (0.101)_2$

只有极少十进制小数有完全相等的二进制小数表示法。大多数情况下，这种转换对应的二进制数是一个无限的小数，请读者自己转换一下十进制 0.8。无论计算多少位，都不会算到整数部分得 0。也就是说，计算机可以精确表示整数，但不能精确表示小数。在程序中写入的 2.8，计算机记录的是稍有误差的一个数，不是精确的 2.8。

因此，不可以用“==”比较两个小数，也不可以用“==”比较一个小数和一个整数。
如：变量 a 是小数，不能写成 `if(a==3)` 或 `if(a==3.0)`，那么怎样进行小数的相等比较呢？

用数学思维中的约等，差不太多就认为相等。如：

`if(a==3)` 应写成 `if(fabs(a-3)<1e-6)`

`if(a==b)` 应写成 `if(fabs(a-b)<1e-6)`

后面的 1e-6 是 0.000001。fabs 是求绝对值的函数。

这个小数的大小应怎样确定，非常小写成 1e-1000 行不行呢？

这个太小了，最小的双精度小数是 1e-308。那就写它了，其他也没必要，只要按数据的实际意义，写一个合理点的就行了。一般情况下常用 1e-6，如果在数学运算中感觉精度不够用，可以再换更小的数。

专家点评

这是一个初学者常犯的错误，尽管大多数情况下这样做也能得出正确结果，但隐患非常大。

问题 96 关系运算符和数学不等号有什么区别？

问题阐述

在编程语言中，关系运算符是最基本的编程元素，它和数学上的不等号用法是否完全



Note



一致呢?

专家解答



每个人都是先读懂程序再学会自己编程序。请看下面这个求解一元二次方程的程序。

Note

```
#include<stdio.h>
#include<math.h>
main()
{
    double a,b,c,x1,x2,p;
    clrscr();
    printf("a*x*x+b*x+c=0");
    printf("input a,b,c,apart with blank:");
    scanf("%lf%lf%lf",&a,&b,&c);
    p=b*b-4*a*c;
    if(p>=0)
    {
        x1=(-b+sqrt(p))/(2*a);
        x2=(-b+sqrt(p))/(2*a);
        printf("x1=%lf\nx2=%lf\n",x1,x2);
    }
}
```

有些初学者看完程序后就会问：“老师，a，b，c 是任意输入的，你怎么知道计算后的 p 就一定大于等于 0？”

关系运算符也叫“比较运算符”。在 C 语言中，关系运算符共有六个，它们是>、>=、<、<=、==、!=。由关系运算符连接的表达式叫关系表达式。“a>b”叫关系表达式。关系表达式的作用是用来比较两个运算量的大小或它们是否相等。注意这里的概念是 a 是否大于 b，分为两种情况，大于怎么办，不大于怎么办。或者说，如果条件成立，程序用什么方式处理，不成立用什么方式处理。

请读者朋友根据上面的分析，为这位同学做一解答。

专家点评

尽管程序设计语言中的关系运算符和数学中的不等号在写法和读法上都很相近，但在应用上，数学上的不等号表示因果关系，程序语言中关系运算符表示假设关系，应用方向截然不同。

问题 97 if 语句后面一定不能写分号吗？

问题阐述

一般来说，if 和 else 语句本身后面不能写分号，但也不是绝对的，请看下例。



```
if(a+b>c&&b+c>a&&c+a>b);  
else printf("重新输入三角形边长");
```

这个程序 if 后面有个分号，这个程序有错误吗？

专家解答

这其实是空语句的问题。

空语句是不执行任何操作的语句。空语句本身是一条语句，语句体内容为空，功能为空。作为语句，其后面要有一个分号，也就是一个空白分号就是一个空语句。

空语句用在程序中从语法格式上来说需要一条语句，但这条语句没有实际功能。程序中没必要出现空语句，如“问题阐述”的例子中，输入三角形边长，如果不能构成三角形，则重新输入边长。

空语句一般出现在选择结构 if 后，表示条件成立什么也不执行，不成立执行 else 下的操作。这时如果条件简单，可以直接写出相对于 else 的条件去构造程序，只有条件比较复杂，写成相对条件更难于理解，才写成这种空语句。

空语句也可能出现在循环 while 后。循环体要执行的功能在循环条件中都已经执行完毕，并且没有可再执行的语句，如果什么也不写，就会导致下一条语句变成循环体，这时用空语句表示。

专家点评

空语句并不常用，它的作用不是不可替代的，所有使用空语句的程序，都可以改成没有空语句的程序。

问题 98 这个程序为什么多执行了好多语句？

问题阐述

有下面一段程序。

```
#include<stdio.h>  
main()  
{  
    int a,b,t;  
    printf("input two number:")  
    scanf("%d%d",&a,&b);  
    if(a>b)  
        t=a;  
        a=b;  
        b=t;  
    printf("%d %d",a,b);  
}
```



Note



运行结果如图 7.2 所示。

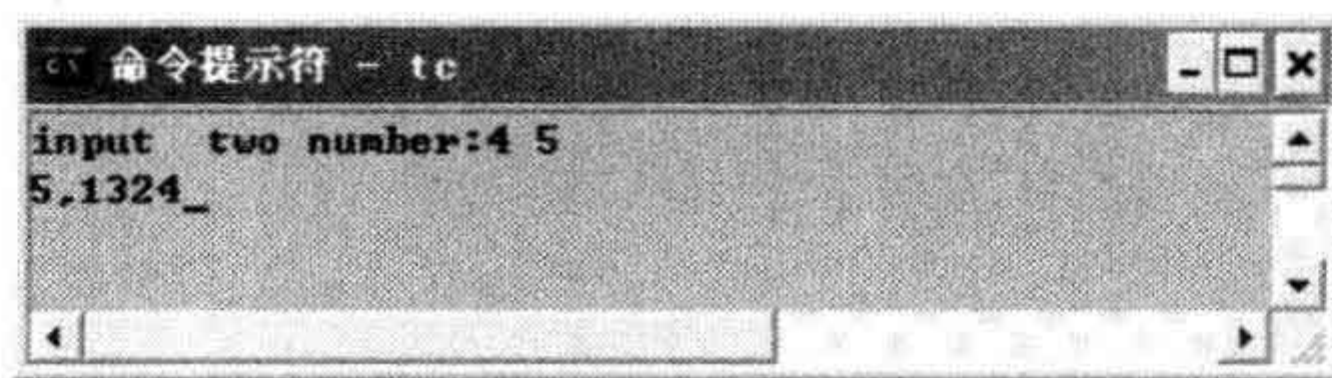


图 7.2 两个数的排序

以上程序的功能是当 a 大于 b 时，交换 a, b 。 a 不大于 b 时，直接输出，即两个数按从小到大的顺序排序。但当程序中的 a, b 输入 4 和 5 时，却没有得出 4 和 5，这是为什么呢？现在的结果好像是多执行了好多语句。

专家解答

C 语言的 `if...else` 语句中，`if` 和 `else` 后面的语句可以是一条，也可以是多条，如果是多条，必须用花括号 “{}” 括起来。花括号括起来的语句叫复合语句。如果只有一条，写不写花括号都可以。

另外，为了使程序逻辑结构清晰，常使用缩进，空行等技巧。

以上程序尽管看起来当 $a > b$ 成立时，执行 “`t=a;a=b;b=t;`”，实际上是 $a > b$ 成立时，执行 “`t=a;`” 而 “`a=b;b=t;`” 任何时候都执行，即程序的逻辑结构实际是：

```
scanf("%d%d",&a,&b);
if(a>b)
    t=a;
a=b;
b=t;
printf("%d,%d",a,b);
```

专家点评

复合语句是由花括号括起来的多条语句。复合语句在执行时认为是一条语句。因此，可以理解为 `if...else` 后任何时候都只能是一条语句。

问题 99 不用关系表达式和逻辑表达式做条件

问题阐述

`if` 语句，`while` 语句后面都要有一个执行的条件，这个条件，多数情况下都是关系表达式（含逻辑表达式），此处可以用关系表达式和逻辑表达式之外的其他表达式形式吗？

专家解答

C 语言没有专门的逻辑值 `true` 和 `false`，一个关系表达式或逻辑表达式的执行结果成立时用 1 表示，不成立时用 0 表示，但不是只有 0 和 1 可以表示逻辑值，而是任何数都可以。



Note



其他的值表示逻辑值时，除了零之外的所有数都是真，整数可以，小数也可以，甚至字符也可以，把 100、3.5 和 ‘a’ 当成逻辑值看待时，都是“真”。

也就是说，if 语句、while 语句后面的那个条件可以是任意类型，当其值为 0 时就是假，不是 0 时就是真。

例 1：求 n!。

```
if(n<=1)
    return 1;
s=1;
while(n)
    s*=n--;
```

例 2：字符串复制。

```
while(*target++=*source++);
```

循环体是个空语句。

例 3：判断一个字符是否为小写字母。

```
#include<ctype.h>
#include<stdio.h>
#define TRUE 1
#define FALSE 0
main()
{
    if(isalpha('a')==TRUE)
        printf("yes");
    else
        printf("no");
}
```

其中的 if(isalpha('a')==TRUE)语句记为 A，下面再写出几种。

B if(isalpha('a')==1)

C if(isalpha('a'))

D if(!(isalpha('a')==TRUE))

E if(isalpha('a')!=FALSE)

这些写法并不完全等价，请考虑哪些是正确的，哪些是错误的。

isalpha 是函数参数不是小写时返回 0，是小写时返回非 0，注意不是 1。因此 AB 都是错误的。CDE 是正确的。

专家点评

有些时候，为程序的执行效率考虑，需要把程序写得结构十分紧凑。不过，如果不是十分看重效率，最好不要用这种写法。只要遇到这种程序，能看懂就行了。“清晰第一，效率第二”是程序设计的基本原则。



Note



问题 100 怎样理解复合语句中的变量?



问题阐述

Note

C 语言程序由很多函数构成, 函数间的相互调用完成程序的整体功能。每个函数由定义部分和执行部分组成。定义部分在前, 或者说一个函数中用到的所有变量都要定义在函数的开始位置。在程序的中间部分可不可以定义变量呢?

专家解答

这种变量就是复合语句中的变量。

复合语句是由花括号括起来的多条语句。在复合语句中也可以定义变量, 这种变量只能存在于复合语句中, 出了复合语句就认为变量没定义。在函数开始处定义的变量称为局部变量。复合语句中定义的变量是比局部变量更“局部”的变量。程序中, 如果局部变量和复合语句变量重名, 编译时并不提示错误, 只是在作用域重叠区域(即复合语句内)使用变量时, 用的才是复合语句变量。例如:

```
#include<stdio.h>
main()
{
    int a,b,c;
    a=1;b=2;c=3;
    if(a<b)
    {
        int c, d;
        c=0;
        d=a+c;
        b=a+c;
        c=a+c;
    }
    printf("a=%d  b=%d c=%d",a,b,c);
}
```

以上程序的输出结果是 a=1, b=1, c=3。

程序中的 c 在复合语句中定义一个, 在整个程序开始时定义一个, 这是两个变量。输出语句中的 c 应该是程序开始的那个, 尽管在复合语句中也有一个值为 1。那 b 怎么不是 2 呢? 因为 b 的值在复合语句中被重新赋值为 1。

专家点评

在复合语句中定义变量并不常用, 尤其在复合语句中定义和局部变量同名的变量更不常用。



问题 101 如何进行 if 语句的嵌套?

问题阐述

有些初学者在进行 if 语句嵌套时,随着层数的增加,越套越乱,那 if 语句的嵌套有没有可供参考的规则呢?

专家解答

有,而且规则很简单。
if 语句的基本结构如下。

```
if(条件)
{
    语句 1
    语句 2
    ...
}
else
{
    语句 3
    语句 4
    ...
}
```

其中 else 及其后面所有语句也可以没有,如果 if 下只有一条语句,这时可以不加花括号,else 下相同。

现在将以上结构中的“语句 1”改成另一条 if 语句(黑体字总部分)。

```
if(条件)
{
    if(条件)
    {
        语句 11
        语句 12
        ...
    }
    else
    {
        语句 13
        语句 14
        ...
    }
}
```



Note



Note

```

    语句 2
    ...
}
else
{
    语句 3
    语句 4
    ...
}

```

语句 2、语句 3 和语句 4 也可以这样替换，或全部替换。

其实语句 11、语句 12、语句 13 和语句 14 也可以再做如上的替换，这样 if 语句的嵌套是任意的，而且其深度也没有明确规定。

专家点评

用一句话来说，就是在可以出现普通语句的地方，也可以出现 if 语句，出现多条语句的地方必须加花括号。

问题 102 条件运算符“?:”怎样应用？

问题阐述

在程序中偶尔可见“?:”运算符，怎样使用这个运算符呢，什么时候使用呢？

专家解答

“?:”叫做条件运算符，它是 C 语言中唯一一个三目运算符，所谓运算符的目，就是一个运算符和几个操作数相结合，例如“a++”，++是单目运算符。“3+2”、“a>b”、+和>都是双目运算符。条件表达式的格式为：

测试表达式?结果表达式 1: 结果表达式 2

整体上叫做一个条件表达式。

条件表达式的值：先求解“测试表达式”的值，如果其值为真，整个表达式的值是“结果表达式 1”，否则是“结果表达式 2”。例如：

```

a=5;
b=4;
c=a>b?a:b;
printf("%d",c);

```

此时 c 的值为 5。

上述 c=a>b?a:b 的功能与



```
if(a>b)
    c=a;
else
    c=b;
```

完全相同。由于条件运算符实现的功能与 if...else 语句完全相同，因此条件运算符在程序中很少用到。

条件运算符的优先级和结合性：

条件运算符的优先级低于逻辑运算符“&&”和“||”，高于赋值运算符“=”，结合性从右向左。

优先级的举例：

```
x=a>=0&&b>=0?a*b:0
```

以上表达式可理解为：

```
x=((a>=0)&&(b>=0)?(a*b):0)
```

结合性的举例：

```
x=a>b&&a>c?a:b>c?b:c
```

由于结合性从右向左，以上表达式可理解为：

```
x=a>b&&a>c?a:(b>c?b:c)
```

在函数调用中，为了程序结构易于理解，条件运算符很少用到。但在宏调用中，由于条件运算符可以实现选择结构的功能，因此得到广泛应用。例如：

```
#define max(a,b) (a)>(b)?(a):(b)
```

专家点评

从上例看，条件运算符写的程序比 if...else 语句要简单得多。因此，两种用法各有各的优点。

问题 103 switch 语句的基本格式是什么？

问题阐述

C 语言中有两个构成选择结构的语句，即构成双分支的 if...else 语句和构成多分支的 switch...case 语句，switch 语句的基本格式是什么？

专家解答

```
switch(表达式)
{
```



Note



Note

```

case 常量表达式 1:
    语句 1
    [break]
...
case 常量表达式 n:
    语句 n
    [break]
[default:
    语句 n+1]
}

```

switch 后的表达式是任意类型的表达式，case 后的常量表达式只能是整型或字符型。它的执行过程是：

先求 switch 后的“表达式”的值。然后在后面的多个 case 中查找此值，如果找到相等的，则执行下面的对应语句，直到遇到一个 break 为止。如果没有 break 的话，将会执行到下一个 case 下的语句。如果 switch 表达式的计算结果与所有 case 均不相等，则执行 default 后的语句。default 也可以没有，即都不相等什么也不执行。

脚下留神：

在 switch 语句中，当找到与 switch 表达式相等的 case 时，执行 case 下的语句。case 下的所有语句都执行完成后，如果一直没有 break，那么程序将会执行到下一个 case，而不管它的值是否与 switch 表达式相等，即多个 case 之间不具有天然的互斥性。要想使程序执行完一个 case 后的语句，而不进入下一个 case，必须使用 break 语句，使程序退出 switch 结构。这样，后面的 case 也就不执行了。

看以下程序的执行结果：

```

#include<stdio.h>
main()
{
    int a,b;
    a=4;
    swich(a%2)
    {
        case 0:
            b=10;
        case 1:
            b=20;
    }
    printf("%d",b);
}

```

以上程序执行的结果是 20。

4%2 结果是 0，应该显示 10 才对呀，怎么会是 20 呢？原因就在于赋值语句 b=10 执



行之后,因为没有 `break` 语句,那就再向下执行,进入 `case 1`,再执行 `b=20`,因此最后的结果显示为 20。要想让它能够显示 10,需要在 `b=10` 的后面, `case 1` 的前面加上一行“`break;`”(包括分号;),这样就可以得出 10 了。

专家点评

`switch` 语句不具备互斥功能,给写程序添加了一个麻烦,即再多写一个 `break`,但它也为多个 `case` 共用一组执行语句提供了条件。



Note

问题 104 if 语句与 switch 语句的优缺点是什么?

问题阐述

`if` 语句和 `switch` 语句都是用来构成选择结构的。`switch` 语句用于构成一个多分支选择结构, `if` 语句用于构成只有两个分支的选择结构,通过 `if` 语句的嵌套,也可以构成多分支选择结构。那么,什么时候使用 `if` 语句的嵌套,什么时候使用 `switch` 结构,二者各有什么优缺点呢?

专家解答

`if` 语句由多个条件构成多分支,而 `switch` 由一个表达式构成多分支,它基于表达式的具体取值而构成多分支。这样看来,可以由 `switch` 构成的多分支结构,必须能表示成由一个表达式的值控制执行流程的形式。对于由不同条件嵌套构成的多分支结构,无法用 `switch` 来表示。

而这种可由一个表达式控制的多分支结构,其表达式的值还必须能表示成有限个离散点,或者取值的一个方向可以为无限,因为此时可以用 `default` 处理。但 `switch` 中只能有一个 `default`,所以它不适合处理两个方向上都是无限值的情况,这时只能借助 `if` 的嵌套来实现。

如果问题性质具有明显的多个离散点,每个点处执行不同处理的特点,这时用 `switch` 语句比用多个 `if` 语句程序结构要清晰很多。

总之, `if` 语句结构灵活,能处理任意多分支; `switch` 结构简单,能处理的问题有限,但用 `switch` 实现的程序逻辑结构清晰。

专家点评

两个结构处理复杂问题时各有千秋,使用时可以灵活应用,但处理简单结构,有 `if...else` 就够用了,不要用 `switch` 去处理双分支,这和用牛刀杀鸡没什么区别。



问题 105 switch 语句中的 default 关键字是否必须?



问题阐述

如题, switch 语句中的 default 关键字是否必须?

专家解答

不是必须的, 可以没有。

default 指前面的 case 都不成立时, 程序要执行的语句都不成立时, 也可以什么都不执行, 这时就没有 default。如同 if...else 语句中的 else 是条件不成立时要执行的语句, 与 else 可以没有是一样的。

switch 语句的格式和执行流程可参见问题 103 (switch 语句的基本格式是什么)。

专家点评

程序控制结构描述的是最完整的结构, 每个程序不一定用到一个结构的所有组成部分。

问题 106 break 关键字在 switch 语句中应注意什么?

问题阐述

switch 语句由 switch 分支点和多个 case 处理入口构成, 每个 case 不具有处理互斥的功能, 一个 case 执行完后, 默认继续执行下一个 case。此时, 如果不希望继续执行下一个 case, 可以用 break 语句退出 switch 结构。break 不是 switch 中的固有组成部分, 那么在 switch 中使用 break 有哪些需要注意的地方呢?

专家解答

(1) 最容易出的问题是, 忘记了 case 后面要写一个 break, 其结果是多个 case 不能互斥, 按顺序执行。例如:

```
scanf("%d",&a);
switch(a%4)
{
case 0:
    printf("A");
case 1:
    printf("B");
case 2:
    printf("C");
```




```
case 3:
    printf("D");
}
```

当程序运行时输入 50，输出的结果是多少？

有些人不假思索就得出 C，其实应该是 CD。

(2) 每个 case 都写 break，其实没有这样的必要。例如，根据考试成绩 0~100 给出对应等级，评级标准为优秀 A：85~100，良好 B：70~84，及格 C：60~69，不及格 D：0~59。代码如下。

```
#include <stdio.h>
main()
{
    int a;
    printf("input examanition score:");
    scanf("%d",&a);
    switch(a/5)
    {
        case 20:
            printf("A");
            break;
        case 19:
            printf("A");
            break;
        case 28:
            printf("A");
            break;
        case 17:
            printf("A");
            break;
        case 16:
            printf("B");
            break;
        case 15:
            printf("B");
            break;
        case 14:
            printf("B");
            break;
        case 13:
            printf("C");
            break;
        case 12:
            printf("C");
            break;
        default:
```



Note



Note

```
        printf("D");  
    }  
}
```

每个 case 后都加 break, 这也是没必要的, 可以把那些相同的结论放在一起, 写成:

```
#include <stdio.h>  
main()  
{  
    int a;  
    printf("input examanition score:");  
    scanf("%d",&a);  
    switch(a/5)  
    {  
        case 20:  
        case 19:  
        case 28:  
        case 17:  
            printf("A");  
            break;  
        case 16:  
        case 15:  
        case 14:  
            printf("B");  
            break;  
        case 13:  
        case 12:  
            printf("C");  
            break;  
        default:  
            printf("D");  
    }  
}
```

专家点评

C 语言中的 switch 语句比较特殊, 一个分支的结束是依赖 break 完成的。case 只决定程序到哪里执行, 而不决定到哪里结束。结束位置由 break 来决定, 没有 break 就一直执行到 switch 完整结构结束。

问题 107 如何正确判断 if 和 else 的匹配?

问题阐述

C 语言中的 if 语句, 后面可以有 else, 也可以没有。在嵌套结构中, 出现一个 else 多



个 if 的情况, 此时的 else 应与哪一 if 匹配呢?

```
b=-1;  
if(a!=0)  
    if(a>0)  
        b=1;  
else  
    b=0;
```

当 $a=0$ 时, b 应该是多少呢?

专家解答

这个问题, 一不留神就会答成得 0, 正确结果应该是 -1。

这是一个嵌套结果中 else 与 if 匹配原则的问题。简单地说, else 与 if 匹配的原则是: else 与和它最邻近的 if 相匹配。

以上例子中的 else 并不与 $\text{if}(a!=0)$ 匹配使用, 而是与 $\text{if}(a>0)$ 匹配使用。以上程序计算机的理解逻辑是:

```
b=-1;  
if(a!=0)  
    if(a>0)  
        b=1;  
    else  
        b=0;
```

因此当 $a=0$ 时, 下面的 if...else 结构整体没执行, 结果 b 为 -1。

程序设计者想的一定是第一种缩进方式的逻辑结构, 认为 a 值为 0 时, b 就应该是 0, 但实际应该是第二种逻辑结构。这是一个十分危险的错误, 用静态走查方法很难发现。由于程序的 3 个分支有两个是正确的, 只有一个不正确, 即使使用通过运行程序的动态测试方法, 如果测试程序时粗心大意, 也很容易把它漏过去。

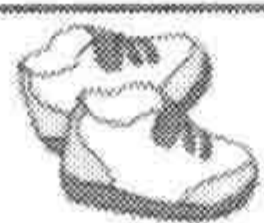
下一个问题是, else 与和它最邻近的 if 相匹配, 如果只想让它与和它不邻近的那个 if 相匹配, 应该怎样表示呢? 很简单, 结构如下。

```
if(条件)  
{  
    if(条件)  
        ...  
}  
else  
{  
    ...  
}
```

这样, else 就与第一个 if 相匹配, 而与最近的那个 if 无关了。



Note



专家点评

以上问题提醒我们，要记住“else 与和它最邻近的 if 相匹配”这一原则。在写程序时，要注意缩进格式，保证缩进格式与逻辑结构是一致的。



Note

问题 108 switch 和 case 后的表达式值的类型是否可以浮点型？

问题阐述

switch...case 语句根据 switch 后面的表达式的计算结果，在后面的多个 case 中找对应相等的值，决定执行对应 case 后的语句。case 后的值不可以是关系表达式，只能是常量或常量表达式。从数据类型上来说，它只能是整型或字符型，不能是浮点型。那么，switch 后面的表达式是否也有此要求，即只能是整型呢？

专家解答

case 后面的值只能是整型，switch 后面的值可以是浮点型，二者在进行值匹配时，先将 switch 计算结果转换为整型，再与 case 进行匹配测试。例如：

$$y = \begin{cases} 2x & x < 20 \\ 3x & 20 \leq x < 30 \\ 4x & 30 \leq x < 40 \\ 5x & 40 \leq x < 50 \end{cases}$$

按数学意义，x 和 y 都可以是小数。C 程序中可以将 x/10，得出一小数，此数值如果得 3、3.5 或 3.51，都执行同一操作 3。程序如下。

```
#include<stdio.h>
main()
{
    double x,y;
    printf("input x");
    scanf("%lf",&x);
    if(x>=50)
        return;
    switch(x/10);
    {
        case 5:
            y=5*x;
            break;
        case 4:
            y=4*x;
```




```
        break;
    case 3:
        y=3*x;
        break;
    default:
        y=2*x;
        break;
}
printf("y=%lf\n",y);
}
```



Note

再说 case 后的表达式，如果把 case 3 改成 case 6/2，此时没有错误，改成 case 7/2 也没有错误。此时，计算机认为是整数 3（两个整数运算的结果是整数），而不是 3.5。如果改成 7/2.0，此时编译错误。

专家点评

switch 后可以不是整型，但与它匹配时会自动转换成整型。case 后必须是整型（含字符型）。

问题 109 区段划分型条件有什么技巧？

问题阐述

区段划分在数学中经常用到，即把整体区间划分成多个小区间，每个小区间都要表示成左端点右端点的形式。在计算机世界，可以使用一些技巧，使区段划分变成只有一个端点。

专家解答

看下面的例子：超市举行积分兑购物券活动，积分满 100 分，可兑 10 元购物券，满 500 分，可兑 60 元购物券，满 1000 分，可兑 150 元购物券，满 5000 分，可兑 1000 元购物券，满 10000 分，可兑 2500 元购物券。

如果使用两个端点的做法。程序可写为：

```
if(score>=100&&score<500)
    money=10;
else if(score>=500&&score<1000)
    money=60;
else if(score>=1000&&score<5000)
    money=150;
else if(score>=5000&&score<=10000)
    money=1000;
```




```
else if(score>=10000)
    money=2500;
```

下面的程序是否正确呢?

```
if(score>=10000)
    money=2500;
else if(score>=5000)
    money=1000;
else if(score>=1000)
    money=150;
else if(score>=500)
    money=60;
else if(score>=100)
    money=10;
```



Note

如果一个人积分 2000 分，他将换得 150 元购物券，程序正确。

这种技巧的特点是先考虑在现实中最难实现的条件，然后逐步到最容易实现的条件。利用 else 语句表示上面条件不成立时执行的功能，实现了区间另一部分的排除，也就实现了另一区间端点的功能。

上面程序，如果把条件从小往大写或者没有顺序，就得不到正确的结果。

专家点评

这种技巧是利用程序执行过程的特点设计的。

问题 110 default 必须写在所有 case 之后吗?

问题阐述

switch...case 语句中用 default 表示 switch 表达式与前面所有 case 均不相等时要执行的语句。那么 default 所在位置，是否必须在所有 case 之后呢？如果可以写在中间，怎样理解 default 的作用呢？

专家解答

不是必须写在所有 case 之后。default 可以出现在任意位置，不在最后时，也表示所有 case 均不相等时执行的语句，包括 default 后面出现的 case。例如：

```
#include <stdio.h>
main()
{
    char a;
    int flag;
    printf("input a character:");
```




Note

```
scanf("%c",&a);
switch(a)
{
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'A':
    case 'E':
    case 'I':
    case 'O':
        flag=1;
        break;
    default:
        flag=0;
        break;
    case 'U':
        flag=1;
}
printf("%d",flag);
}
```

运行结果如图 7.3 所示。

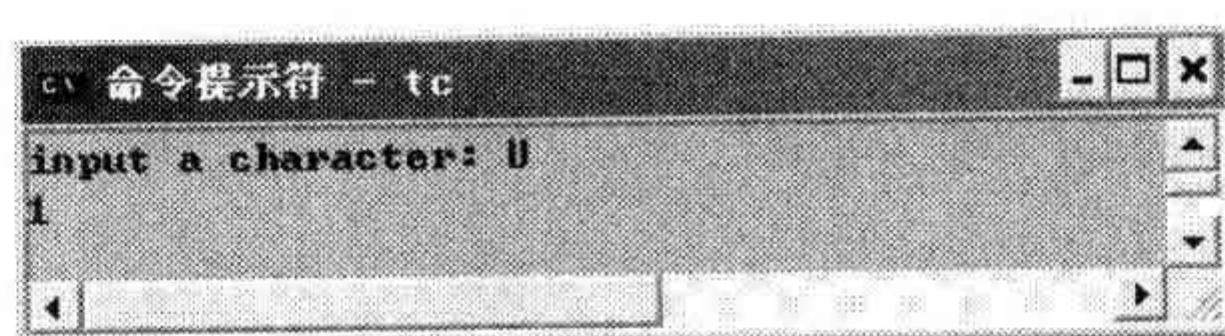


图 7.3 default 写在程序中间的运行结果

以上程序执行时输入 U，输出结果是 1。不会因为 default 在 U 前，就把“aeiouAEIO”之外的都归入到 default 中，即与它出现的位置无关，default 表示所有 case 都不相等时执行的操作。

专家点评

尽管 default 与位置无关，但写程序时，最好写在最后，以增强程序的可读性。

第 8 章

循环结构

- ▶▶ 循环结构的基本概念是什么？
- ▶▶ while 语句的基本格式是什么？
- ▶▶ while 循环应注意什么问题？
- ▶▶ for 循环语句的基本格式是什么？
- ▶▶ for 语句的三个表达式都是必须的吗？
- ▶▶ do...while 语句的基本格式是什么？
- ▶▶ 分号在循环体中的作用？
- ▶▶ while 与 do...while 的区别？
- ▶▶ 什么是循环嵌套？
- ▶▶ 循环嵌套的结构是怎样的？
- ▶▶ 如何正确使用循环嵌套？
- ▶▶ 死循环是怎样产生的？
- ▶▶ 怎样提高循环语句的效率？
- ▶▶ continue 语句的基本作用是什么？
- ▶▶ break 语句的基本作用是什么？
- ▶▶ goto 语句的基本格式是什么？如何使用？
- ▶▶ goto 语句的缺陷是什么？
- ▶▶ 如何选择循环语句？
- ▶▶ 如何判定循环结束和提前结束？
- ▶▶ 如何避免循环中的初值错误问题？



问题 111 循环结构的基本概念是什么？

问题阐述

在实际问题中，经常会用到循环结构，如求 100 以内的 n 的阶乘、杨辉三角等，那什么是循环结构呢？

专家解答

循环结构也就是反复执行一段指令，直到满足某个条件为止。例如，要计算一个公司的所有消费总额，就要将所有的消费加起来。相同的事物从不同的角度理解，就可以得到不同的结论。循环结构还可以理解为：在给定的条件成立时，反复地执行相应的程序，直到给定的条件不成立为止。

循环结构流程如图 8.1 所示。

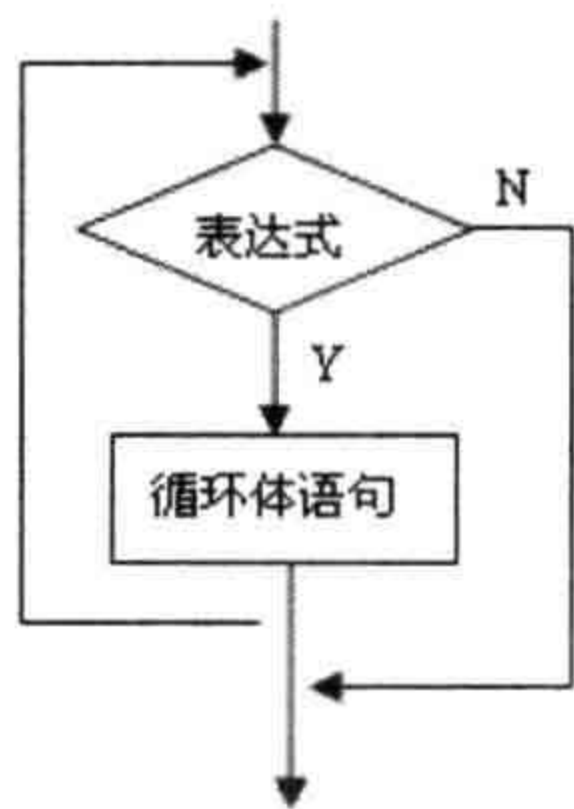


图 8.1 循环结构流程图

例如，要计算 100 以内所有整数的总和。可以认为这个循环程序是反复执行 100 以内整数相加，直到满足求出 100 以内的整数的总和时程序结束；也可以认为从 0 开始，一直与比自己大 1 的整数相加，累积求和，直到加到 100，到 101 时则超出给定的 100 以内整数的条件，程序停止。

这种重复的过程就称为循环，给定的条件就是循环条件，反复执行的相应程序为循环体。使用循环语句可以避免大量重复不必要的操作，能够让程序得到简化，节约内存，提高效率等。

循环语句也可以分为两大类：一类是入口循环语句，一类是结束条件循环语句。入口循环语句是先判断循环条件再循环；结束条件循环语句是先执行一次循环体，再判断循环条件。

专家点评

C 语言中有三种循环语句：`while`、`do...while` 和 `for` 循环语句。循环结构是结构化程序设计的基本结构之一。因此，熟练掌握循环结构是程序设计的基本要求。



Note



问题 112 while 语句的基本格式是什么？

问题阐述

C 语言中有三种循环语句，while 语句是其中的一个，它的基本格式是怎样的呢？

专家解答

while 语句的一般形式为：

```
while(表达式)
    语句;
```

其中，表达式是循环条件，语句为循环体。

注意：

while 是 C 语言中的关键字。while (表达式) 中的表达式，可以是 C 语言中任何符合标准的表达式，而且循环体是否会得到执行，全由表达式决定。

其语句执行流程图如图 8.2 所示。

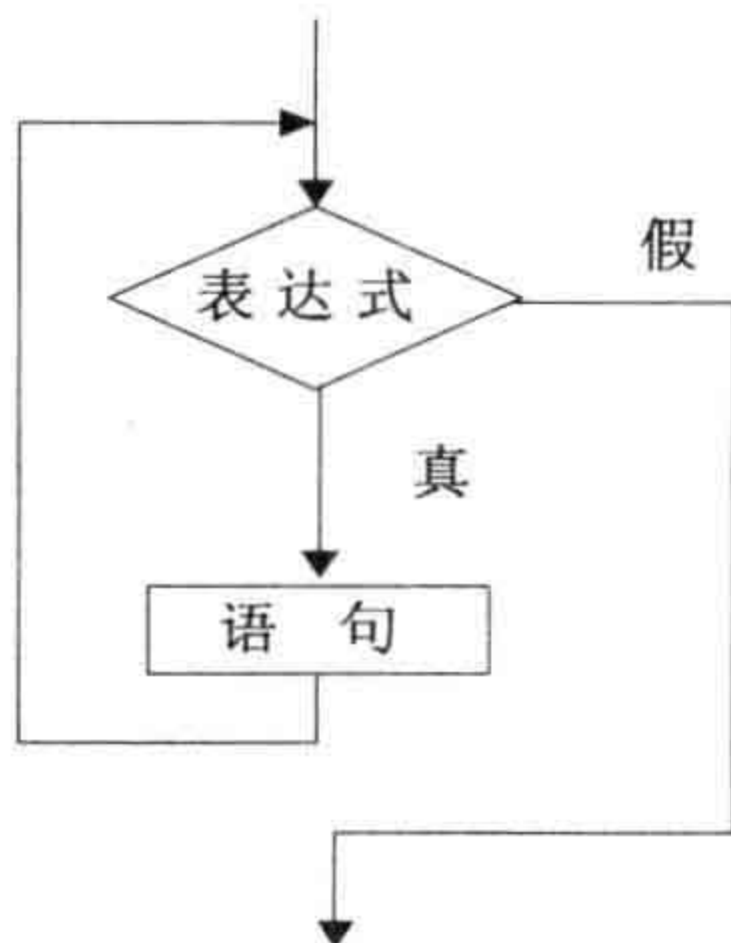


图 8.2 while 语句执行流程

while 语句首先检验一个条件，也就是括号中的表达式。当条件为真时，就执行紧跟其后的语句或者语句块。每执行一遍循环，程序都将回到 while 语句处，重新检验条件是否满足。如果一开始条件就不满足的话，则跳过循环体里的语句，直接执行后面的程序代码。如果第一次检验时条件满足，那么在第一次或其后的循环过程中，必须有使得条件为假的操作，否则，循环无法终止。

例如，统计从键盘输入一行字符的个数，代码如下：

```
#include<stdio.h>
void main()
{
    int n=0;
```




```
printf("input a string:\n");
while(getchar()!='\n') n++;
printf("%d",n);
}
```

程序运行结果如图 8.3 所示。

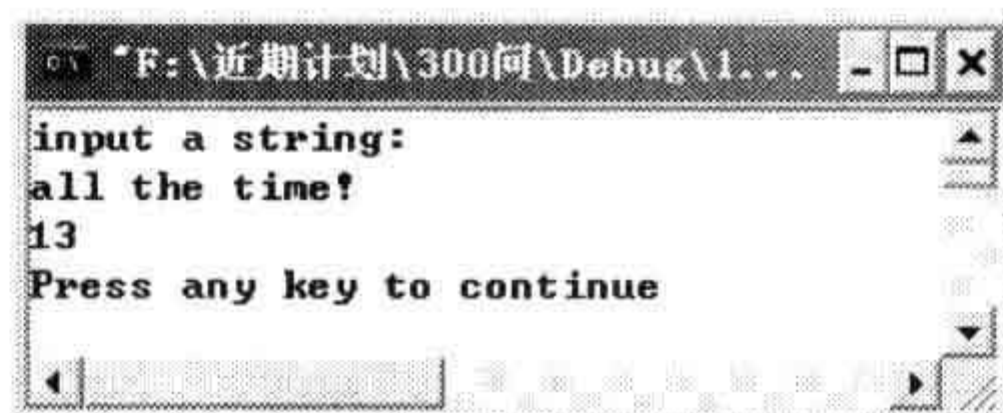


图 8.3 统计字符个数

本例程序中的循环条件为 `getchar()!='\n'`，其意义是，只要从键盘输入的字符不是回车，就继续循环。循环体 `n++` 完成对输入字符个数的计数，从而实现了输入一行字符的字符个数计数。

专家点评

`while` 语句的意思就是当……（表达式）就……（语句），当符合条件时，就会执行 `while` 后面的语句；当不符合条件时，便不会执行 `while` 语句。书写时要注意其格式，不要在 `while` 后加分号，以免造成错误。

问题 113 while 循环应注意什么问题？

问题阐述

`while` 语句的特点是先判断条件，后执行语句，运用好 `while` 语句可以使程序更简捷，那么在使用的时候应该注意什么问题呢？

专家解答

使用 `while` 语句应注意以下几点：

(1) `while` 语句中的表达式一般是关系表达式或逻辑表达式，只要表达式的值为真（非 0）即可继续循环。

```
void main()
{
    int a=0,n;
    printf("\n input n: ");
    scanf("%d",&n);
    while(n--)
        printf("%d ",a++*2);
}
```



Note



本程序将执行 n 次循环，每执行一次， n 值减 1。循环体输出表达式 $a++*2$ 的值。该表达式等效于 $(a*2; a++)$ 。

(2) 循环体若包括一个以上的语句，则必须用 $\{$ 括起来，组成复合语句。如果忽略了花括号的用途，语句块没有用花括号括起来，则 `while` 语句的范围只到 `while` 后面的第一个分号处结束，程序就会得不到预计的结果。

(3) 应注意循环条件的选择，以避免死循环。

```
void main()
{
    int a,n=0;
    while(a=5)
        printf("%d ",n++);
}
```

本例中，`while` 语句的循环条件为赋值表达式 $a=5$ ，因此该表达式的值永远为真，而循环体中又没有其他终止循环的手段，因此该循环将无休止地进行下去，形成死循环。

(4) 允许 `while` 语句的循环体又是 `while` 语句，从而形成双重循环。

专家点评

`while` 循环语句因为它的一些特性，常常会让初学者马虎大意，出现细节上的错误，从而不能得到预期的结果。为了避免此类现象的发生，应注意 `while` 语句的使用。

问题 114 for 循环语句的基本格式是什么？

问题阐述

`for` 循环语句在 C 语言中是最为常见的循环语句，其功能强大，而且用法灵活，那么它的基本格式是什么呢？

专家解答

`for` 语句的一般形式为：

```
for(表达式 1; 表达式 2; 表达式 3)
    语句;
```

每条 `for` 语句包含三个用分号隔开的表达式，这三个表达式可以是任意形式的表达式，大多数情况下都是用于对 `for` 循环的控制。`for` 循环语句和 `while` 循环语句在语法上要求有一点相同，就是若在循环体内需要多条语句进行描述时，必须用花括号将多条语句括在一起，形成一条复合语句。

其语句执行流程图如图 8.4 所示。



Note

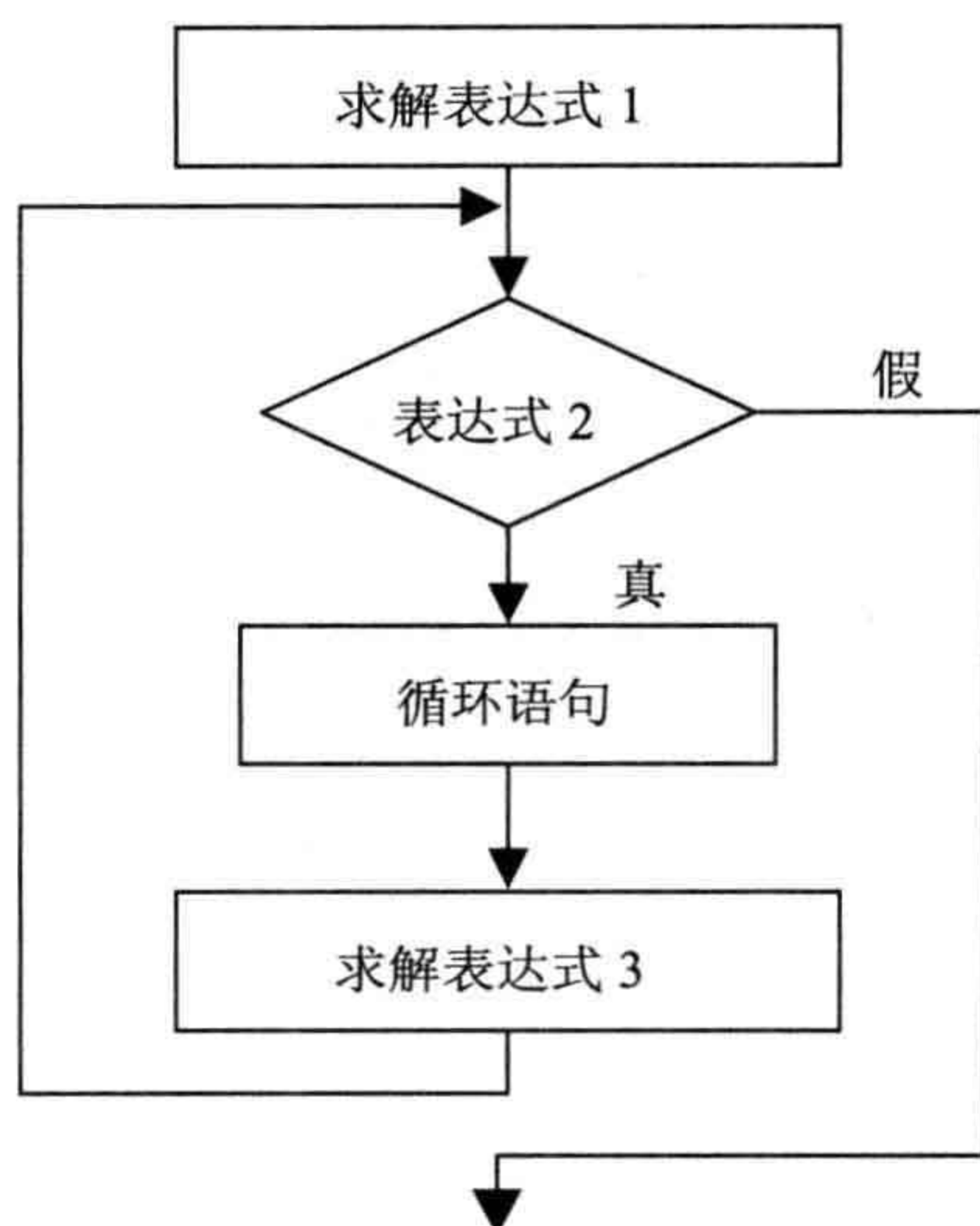


图 8.4 for 语句执行流程

根据流程图的显示，在 for 循环中，语句的执行过程如下：

- (1) 先计算表达式 1 的值；
- (2) 然后计算表达式 2 的值，如果表达式 2 为真（非 0），则执行一次循环体；否则，跳出循环，执行步骤 5；
- (3) 计算表达式 3 的值；
- (4) 跳转回第 2 步，循环执行；
- (5) 循环结束，执行 for 循环下面的相应语句。

其实，for 语句简单的应用形式如下：

```
for(循环变量赋初值;循环条件;循环变量)
    语句块;
```

for 循环三个表达式和语句块的作用如下：

表达式 1：用来完成变量的初始化，一般是一个赋值表达式，是用来控制循环的变量，所以称之为循环变量，表达式 1 称之为循环变量赋初值。

表达式 2：其作用主要是进行判断，表达式 2 的作用和 while 循环语句中表达式的作用大致相同。若判断其值为真（非 0），则执行循环体；否则，跳出循环体。每次执行完循环体都会再次判断表达式 2 的值，用来决定是否再次执行循环。

表达式 3：其主要功能为修改变量的值，使变量的值做出相应的改变，从而使循环程序逐渐接近结束条件。每执行一次循环体，都会相应地执行此表达式，对变量做出相应的修改。

语句块：前面所说的循环体其实就是语句块，循环体可以是一条语句，或者是一条复合语句，最应该注意的就是复合语句的花括号“{ }”问题。

例如，实现一个循环操作。



Note

```
for(i=1;i<100;i++)
{
    printf("the i is:%d",i);
}
```

在上面的代码中，表达式 1 处是对循环变量 *i* 进行赋值操作，然后表达式 2 处是进行判断循环条件是否为真。因为 *i* 的初值为 1，所以小于 100，执行语句块中的内容。第三个变量是每一个次循环后，对循环变量的操作，然后再判断表达式 2 处的状态。为真时，继续执行语句块；为假时，循环结束，执行后面的程序代码。

专家点评

通过了解 for 循环，对 for 循环有了基本的认识。for 循环是 C 语言中最灵活的循环语句，它可以把循环体和一些与循环控制无关的操作也作为表达式 1 或表达式 3 出现，从而使得程序更简洁。

问题 115 for 语句的三个表达式都是必须的吗？

问题阐述

for 语句中有三个表达式，表达式 1 通常是用来为循环变量赋初值的，表达式 2 通常是控制循环的条件，表达式 3 通常可以用来修改循环变量的值，这三个表达式是缺一不可的吗？

专家解答

在程序的编写过程中，这 3 个表达式可以根据情况进行省略，有以下不同的情况：

(1) for 语句中省略表达式 1。for 语句中，第一个表达式的作用是对循环变量设置初值。因此，如果省略了表达式 1，就会跳过这一步操作，则应在 for 语句之前给循环变量赋值。例如：

```
for(i<10;i++)
```

注意：

省略表达式 1 时，其后的分号不能省略。当表达式 1 省略时，一般都需要在 for 循环语句之前给变量赋初值。

例如，实现 1 到 100 数字间的累加计算，省略 for 语句中的第一个表达式，代码如下。

```
#include<stdio.h>
int main()
{
    int i=1;                                     /*定义变量，为变量赋初始值*/
```




```

int iSum=0;                                /*保存计算后的结果*/
/*使用 for 循环*/
for(;i <=100;i++)
{
    iSum=i +iSum;                          /*累加计算*/
}
printf("1 到 100 数字间的累加计算:%d\n",iSum); /*输出计算结果*/
return 0;
}

```



Note

从代码中可以看到 for 语句中将第一个表达式省略，而在定义 i 变量时直接为其赋初值。这样在使用 for 语句循环的时候就不用为 i 赋初值，从而省略了第一个表达式。

程序运行结果如图 8.5 所示。

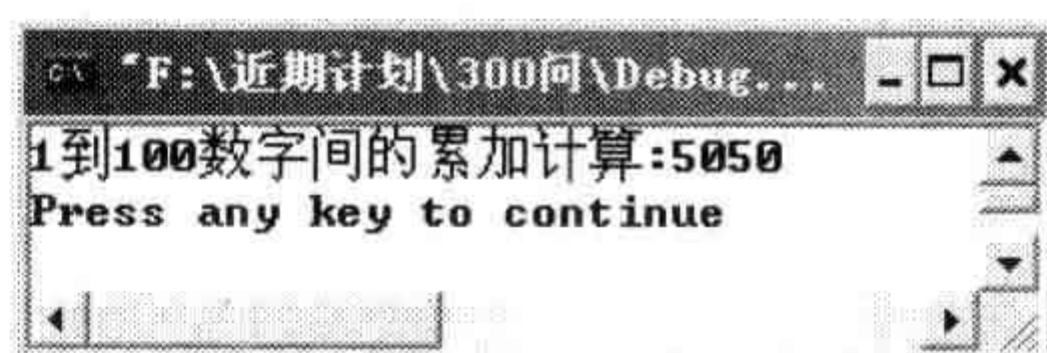


图 8.5 省略 for 语句中的第一个表达式

(2) for 语句中省略表达式 2。如果表达式 2 省略，即不判断循环条件，循环将无终止地进行下去，也就是默认表达式 2 始终为真。例如：

```

for(iCount=1; ;iCount++)
{
    sum=sum+iCount;
}

```

在括号中，表达式 1 为赋值表达式，而表达式 2 是空缺的，这样就相当于使用 while 语句。

```

iCount=1;
while(1)
{
    sum=sum+iCount;
    iCount++;
}

```

注意：

如果表达式 2 是空缺的，将会无限循环。

(3) for 语句中省略表达式 3。表达式 3 也可以省略，但此时程序设计人员应该另外设法保证循环能正常结束，否则程序会无终止地循环下去。例如：

```

for(iCount=1;iCount<50;)
{
    sum=sum+iCount;
}

```




Note

```
iCount++;
```

```
}
```

(4) 3 个表达式都省略。这种情况既不设置初值,也不判断条件,也没有改变循环变量的操作。程序会无终止地执行循环体,如:

```
for(;;)
```

```
{
```

```
    语句
```

```
}
```

这相当于 while 永远为真的情况:

```
while(1)
```

```
{
```

```
    语句
```

```
}
```

专家点评

在 C 语言中,for 循环语句使用最为灵活,其语句中的三个表达式不都是必须的,根据不同的情况可以进行不同的省略,但无论哪种情况,省略时都应注意分号不可以省略。

问题 116 do...while 语句的基本格式是什么?

问题阐述

C 语言中有三种循环语句,do...while 语句是其中的一个,它的基本格式是怎样的呢?

专家解答

do...while 语句的一般形式为:

```
do
```

```
    语句;
```

```
while(表达式);
```

其中语句是循环体,表达式是循环条件。

其语句执行流程如图 8.6 所示。

do...while 语句是这样执行的:首先执行一次循环体语句中的内容,然后判断表达式。当表达式的值为真时,返回重新执行循环体语句。循环执行,直到表达式的判断为假时止,此时循环结束。



Note

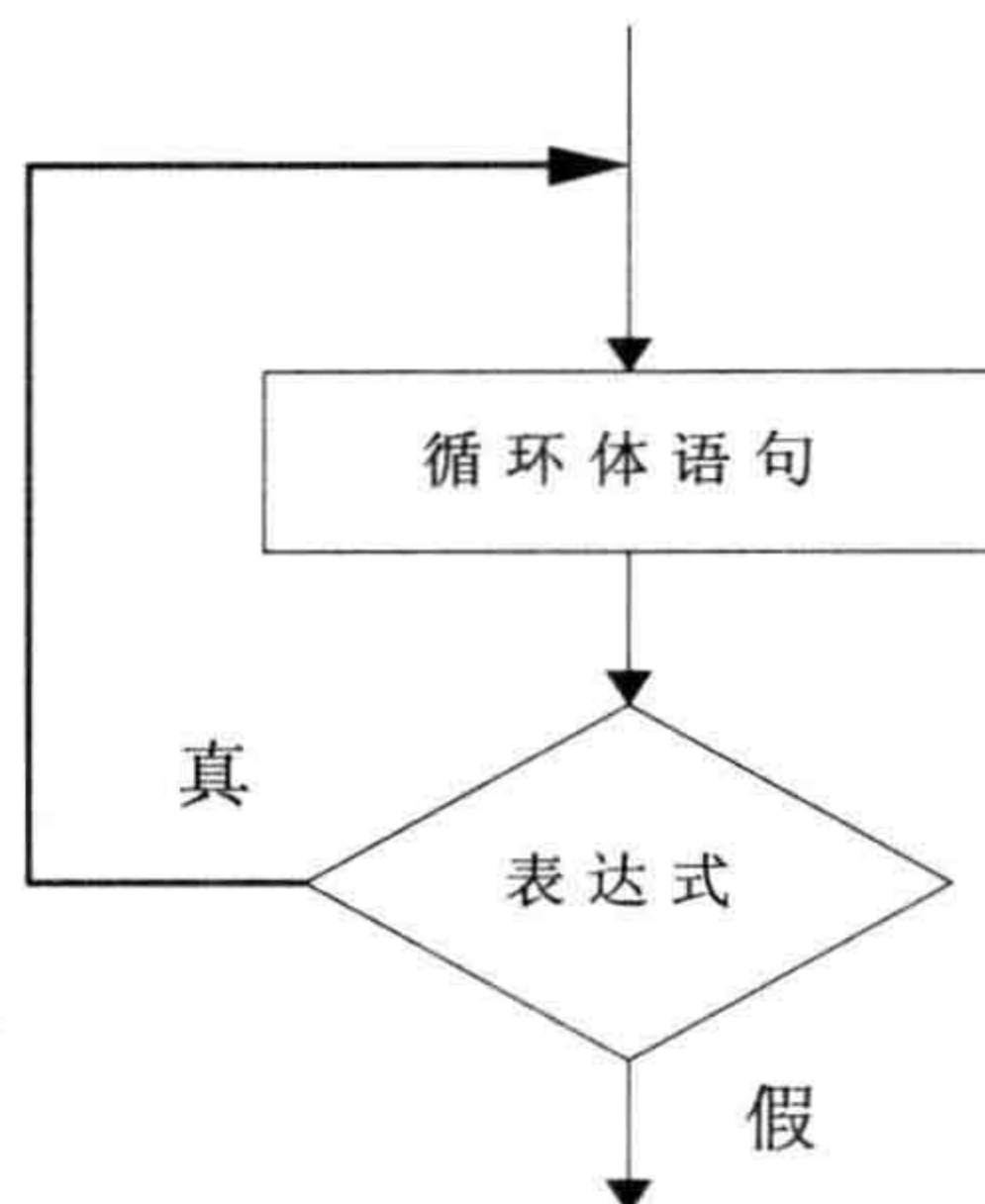


图 8.6 do...while 语句执行流程

例如：

```

do
{
    i++;
}
while(i<100);
  
```

在上面的代码中，首先执行 `i++` 的操作，也就是不管 `i` 是否小于 100 都会执行一次循环体中的内容。然后判断 `while` 后括号中的内容，如果 `i` 小于 100，则再次执行循环语句块中的内容。为假条件时，执行下面的程序操作。

注意：

在使用 `do...while` 语句时，条件要放在 `while` 关键字后面的括号里，最后必须加上一个分号。

`while` 语句和 `do...while` 语句一般都可以相互改写。例如下面的代码。

```

void main()
{
    int a=0,n;
    printf("\n input n:");
    scanf("%d",&n);
    while (n--)
        printf("%d ",a++*2);
}
  
```

转换成 `do...while` 的形式是：

```

void main()
{
    int a=0,n;
    printf("\n input n: ");
  
```




```
scanf("%d",&n);
do
    printf("%d ",a++*2);
while (--n);
}
```

从代码中可以看出，只是将循环条件改变了，由 `n--` 改为 `--n`，如不改则多执行一次循环，这是由 `do...while` 的特点——先执行后判断造成的。

专家点评

从上面的内容可以了解到 `do...while` 的基本格式和含义，但对于 `do...while` 语句还应注意以下几点：

- (1) `do...while` 语句也可以组成多重循环，而且也可以和 `while` 语句相互嵌套。
- (2) 在 `do` 和 `while` 之间的循环体由多个语句组成时，也必须用 `{}` 括起来，组成一个复合语句。
- (3) `do...while` 和 `while` 语句相互替换时，要注意修改循环控制条件。先执行一次循环体，再判断条件。

问题 117 分号在循环体中的作用？

问题阐述

在 C 语言中，分号“`;`”用于结束一个语句，就如同平日常用的句号“`.`”一样，但不是所有的分号都是如此，循环中就不完全是这样，那么分号在循环体中有什么作用呢？

专家解答

(1) `while` 循环与分号。`while`（表达式）后面是没有分号的，如果不小心加上分号，系统也不会出现编译错误，但是程序会不停地执行空操作，形成空循环体，无法执行“`while`（表达式）；”后面的程序，得不到预计的效果。例如，在累加求和的程序中，`while` 循环加上分号产生错误。代码如下：

```
#include<stdio.h>
int main()
{
    int n,i,iSum;                                /*定义三个整型变量*/
    iSum=0;                                       /*给变量赋值*/
    printf("请输入一个整数:");                  /*输出提示信息*/
    scanf("%d",&n);                               /*要求输入一个数值*/
    i=n;
    while(i);                                    /*使用 while 循环*/
}
```





```

    iSum+=i--;
    printf("计算%d 以内的整数总和为: %d\n",n,iSum);
    return 0;
}

```

/*进行运算*/
/*将结果输出*/

程序运行结果如图 8.7 所示。

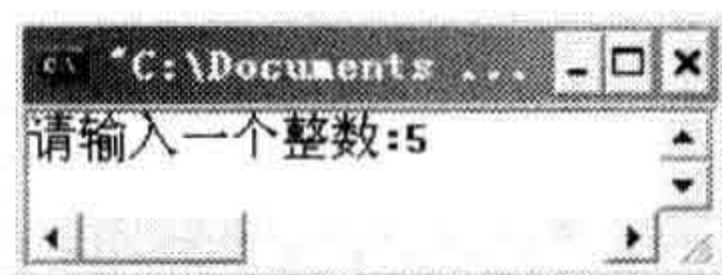


图 8.7 加上分号的错误程序

实际上，例子中的

```
while(i);
```

等价于

```
while(i)
{

```

(2) do...while 循环与分号。在 do...while 循环语句中，while 后面是有分号的，与普通的语句一样，这个分号是用来结束语句的。如果遗漏了这个分号，程序不会通过编译并且产生错误。因此，在用 do...while 循环语句时，不要忘记在 while 的后面加上分号。即使忘记了，编译器也会有所提示，而不会产生不可预见的错误。

注意：

分号在 do...while 语句中代表一条语句的结束，并不代表循环体的结束。

(3) for 循环与分号。for 语句中的各表达式是可以省略的，但唯独不能省略的就是间隔符分号。如下面的例子。

```

#include<stdio.h>
void main()
{
    int n=0;
    printf("input a string:\n");
    for(;getchar()!='\n';n++);
    printf("%d",n);
}

```

程序运行结果如图 8.8 所示。

本例中省去了 for 语句的表达式 1，表达式 3 也不是用来修改循环变量的，而是用作输入字符的计数。这时，循环体是空语句，空语句后的分号不可少，如缺少此分号，则把后面的 printf 语句当成循环体来执行。其运行结果如图 8.9 所示。



Note



Note

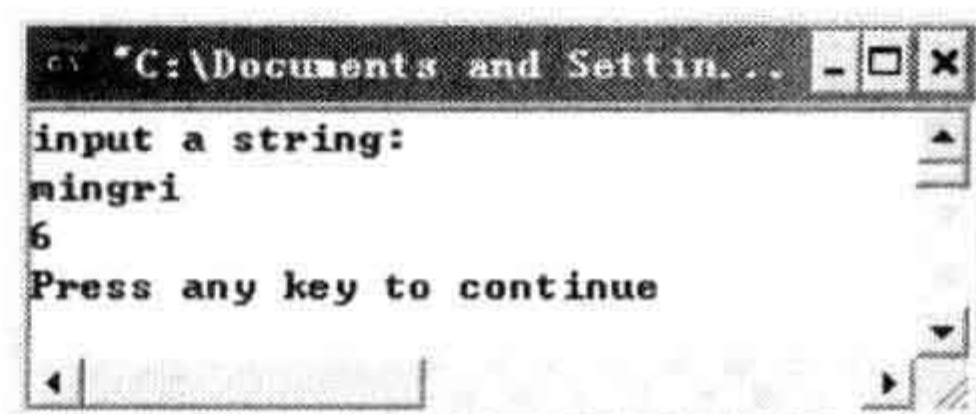


图 8.8 for 语句后加分号

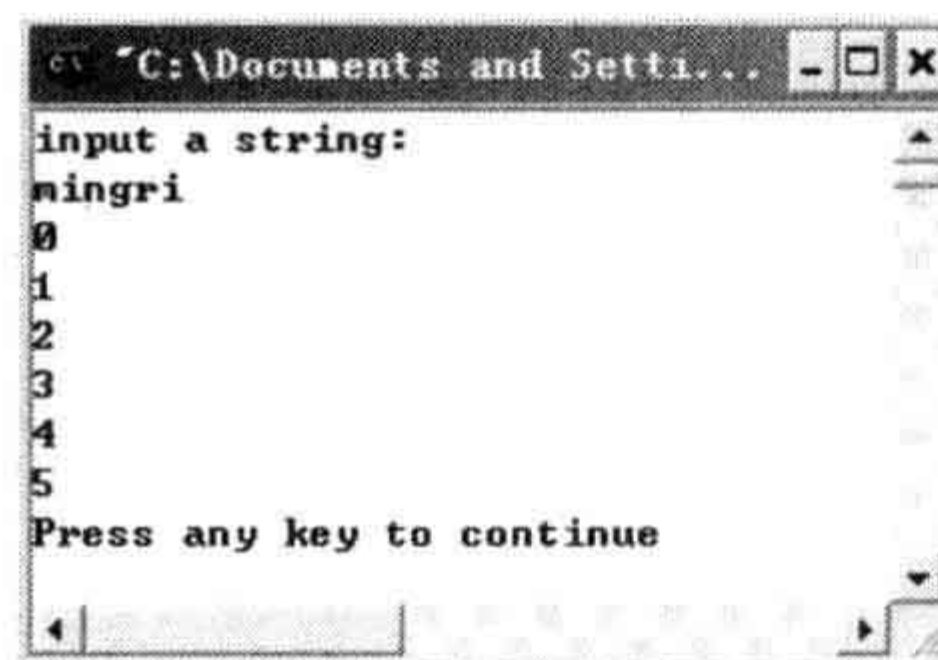


图 8.9 for 语句后不加分号

注意:

如循环体不为空语句时, 绝不能在表达式的括号后加分号, 否则程序会认为循环体是空语句而不能反复执行。

专家点评

在 if 语句和 while 语句中, 表达式后面都不能加分号表示语句的结束, 而在 do...while 语句的表达式后面则必须加分号来表示语句的结束。在 for 循环中, 加分号和不加分号的意义完全不同, 所以一定要注意分号在循环体中的使用。

问题 118 while 与 do...while 的区别?**问题阐述**

while 语句和 do...while 语句类似, 都是要判断循环条件是否为真。如果为真, 则执行循环体, 否则退出循环。它们之间有什么区别呢?

专家解答

while 语句和 do...while 语句的区别在于: do...while 语句是先执行一次循环体, 然后再判断。因此 do...while 语句至少要执行一次循环体。而 while 是先判断后执行, 如果条件不成立或不满足, 则一次循环体也不执行。

下面通过两个例子来区分二者。

while 循环:

```
#include<stdio.h>
main()
{
    int i=15,sum=0;
    while(i<15)
        sum=sum+i;
    printf("the sum is:%d\n",sum);
}
```

/*求和*/
/*将所求结果输出*/

程序运行结果如图 8.10 所示。

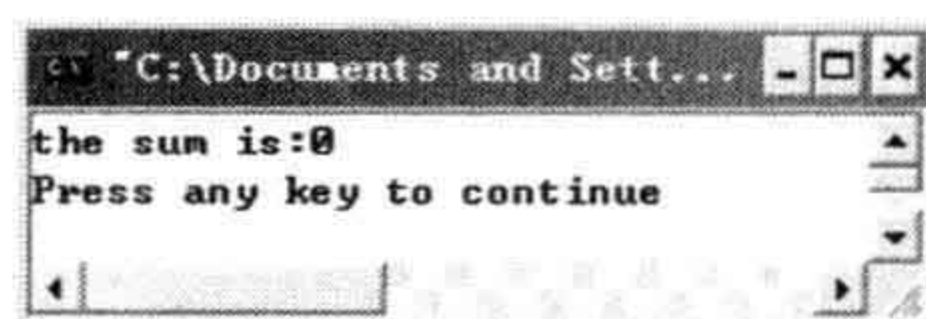


图 8.10 while 求和

do...while 循环:

```
#include<stdio.h>
main()
{
    int i=15,sum=0;
    do                                     /*使用 do...while 判断*/
    {                                     /*循环体语句*/
        sum=sum+i;
    }while(i<15);
    printf("the sum is:%d\n",sum);
}
```

程序运行结果如图 8.11 所示。

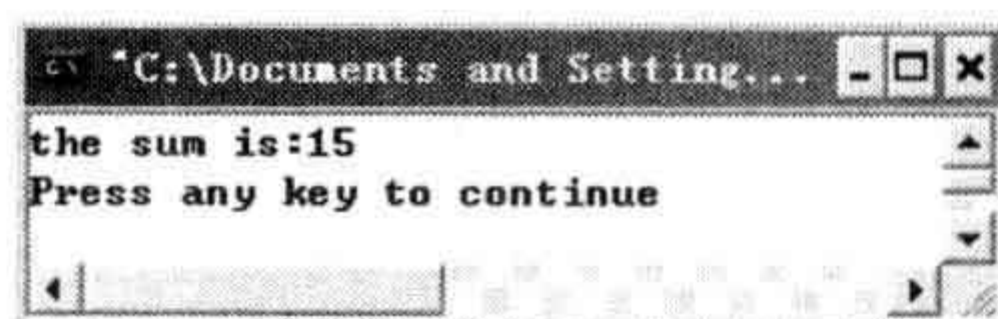


图 8.11 do...while 求和

上面两个例子将 while 语句换成了 do...while 语句，程序运行出的结果就截然不同，主要是因为 while 语句先判断后执行，先判断 $i < 15$ 表达式是否为真，因为表达式为假，故没有执行循环体语句；而 do...while 是先执行后判断，无论表达式是否为真都先执行，执行完再判断。

说明:

while 语句和 do...while 语句一般都可以互相改写，但是 do...while 语句和 while 语句在相互替换时，要注意修改循环控制条件。

专家点评

do...while 语句和 while 语句的区别是 do...while 是先执行后判断，因此 do...while 至少要执行一次循环体。而 while 是先判断后执行，如果条件不满足，则一次循环体语句也不执行。掌握了二者的区别，便可以针对不同的问题使用不同的循环结构。

问题 119 什么是循环嵌套?

问题阐述

分支结构是可以进行嵌套的，循环结构同样也支持嵌套，那什么是循环嵌套呢？





专家解答

一个循环体内又包含另一个完整的循环结构，就称之为循环嵌套。内嵌的循环中还可以嵌套循环，这就是多层循环，也叫做多重循环。

下面通过一些例子来介绍循环嵌套。

使用嵌套语句输出金字塔，代码如下。

```
#include<stdio.h>
int main()
{
    int i, j, k;                /*定义变量 i, j, k 为基本整型*/
    for (i = 1; i <= 5; i++)    /*控制行数*/
    {
        for (j = 1; j <= 5-i; j++) /*空格数*/
            printf(" ");
        for (k = 1; k <= 2 * i - 1; k++) /*显示*号的数量*/
            printf("*");
        printf("\n");
    }
    return 0;
}
```

本例利用了 for 循环的嵌套。首先通过外层循环控制三角形的行数，也就是三角形的高度。然后在循环中嵌套循环语句，也就是内层循环，控制每一行输出的空白和*号的数量。这样就可以将整个金字塔的形状进行输出，如图 8.12 所示。

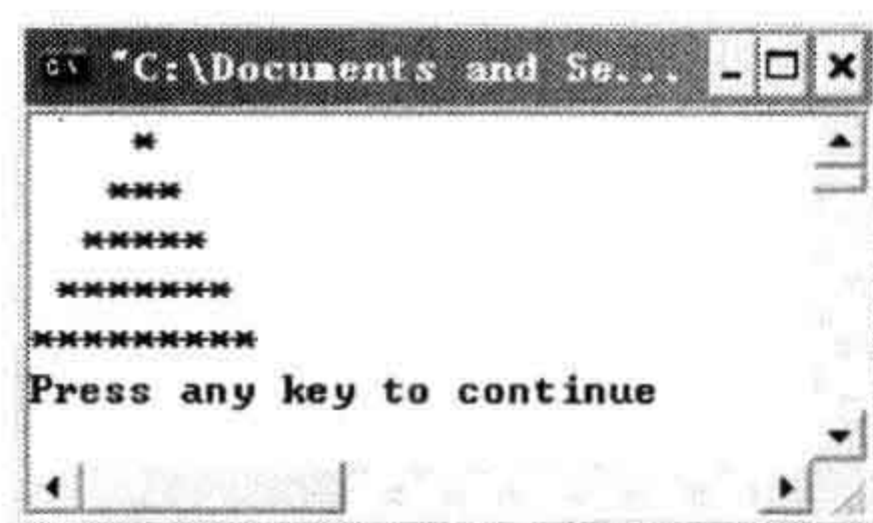


图 8.12 金字塔

使用循环嵌套打印杨辉三角，代码如下。

```
#include<stdio.h>
void main()
{
    int i, j, a[11][11];        /*定义 i, j, a[11][11]为基本整型*/
    for (i = 1; i < 11; i++)    /*for 循环 i 的范围从 1 到 10*/
    {
        a[i][i] = 1;            /*对角线元素全为 1*/
        a[i][1] = 1;            /*每行第一列元素全为 1*/
    }
    for (i = 3; i < 11; i++)    /*for 循环范围从第 3 行开始到第 10 行*/
```





```

        for (j = 2; j <= i - 1; j++)          /*for 循环范围从第 2 列开始到该行行数减一列为止*/
            a[i][j] = a[i - 1][j - 1] + a[i - 1][j]; /*第 i 行 j 列等于第 i-1 行 j-1 列的值加上第 i-1 行
                                                    j 列的值*/

    for (i = 1; i < 11; i++)
    {
        for (j = 1; j <= i; j++)
            printf("%4d", a[i][j]);          /*通过上面两次 for 循环将二维数组 a 中元素输出*/
        printf("\n");                       /*每输出完一行进行一次换行*/
    }
}

```



Note

杨辉三角中数字间有规律：每一行的第一列均为 1；对角线上的数字也均为 1；除每一行第一列和对角线上数字外，其余数字均等于其上一行同列数字与其上一行前一列数字之和，如图 8.13 所示。

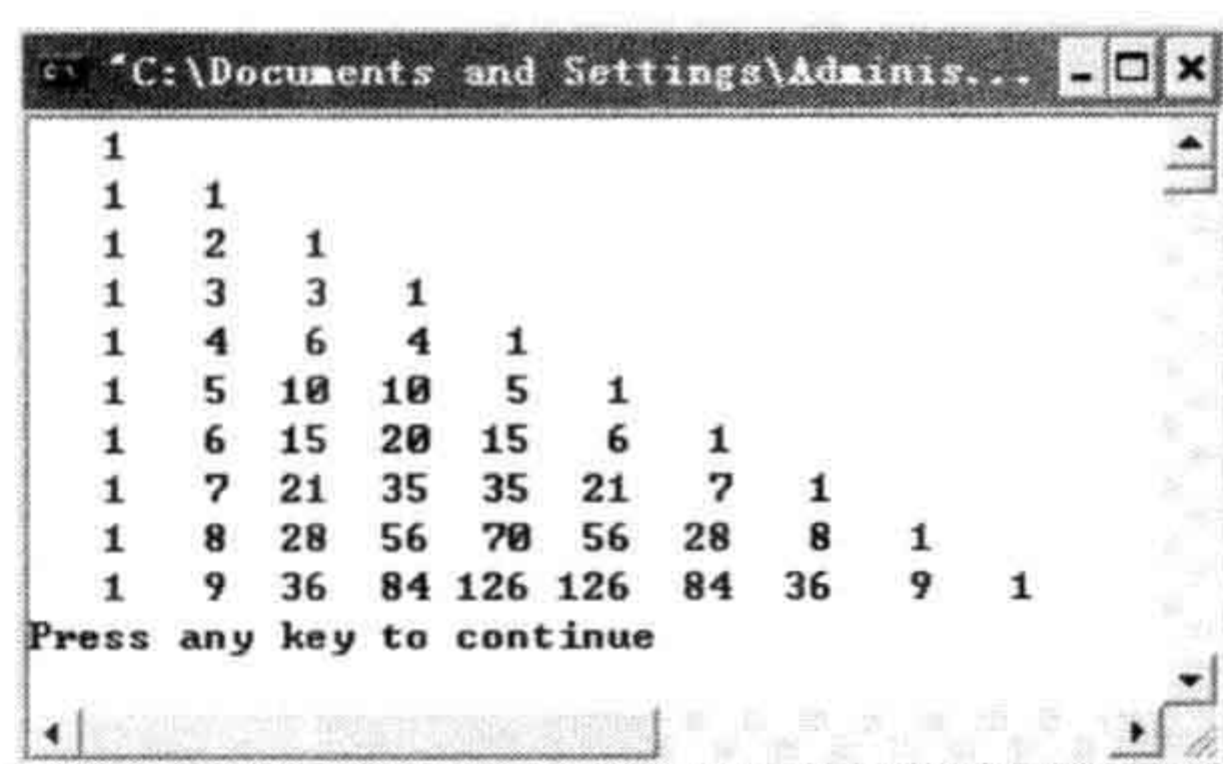


图 8.13 杨辉三角

程序的第一个 for 循环中，变量 i 的范围从 1 到 10，循环体中语句 a[i][i] 将对角线元素置 1，语句 a[i][1]=1 将每行中的第一列置 1。用 for 循环嵌套实现除对角线和每行第一个元素外其他元素的赋值过程，即 a[i][j]=a[i-1][j-1]+a[i-1][j]，再次使用 for 循环的嵌套将数组 a 中的所有元素输出。

以上就是 for 循环的嵌套。利用循环嵌套一般可以达到在某一范围内穷举的目的。

注意：

应用循环嵌套时，不要在循环体内改变循环变量的值，否则会导致整个循环出现问题。

专家点评

循环嵌套从总体来说，其实就是一个循环语句，只是循环体内又构成了另一个循环。正因如此，循环嵌套最适宜描述一些特定的算法，如乘法口诀的输出，百元买百鸡等等。

问题 120 循环嵌套的结构是怎样的？

问题阐述

循环语句有 while 语句、do...while 语句和 for 语句这三种，它们之间是否可以嵌套呢？如果可以，其结构是什么样的呢？



专家解答

三种循环（即 while 循环、do...while 循环和 for 循环）之间可以互相嵌套。例如，下面几种嵌套方式都是正确的。

(1) while 结构中嵌套 while 结构。

```
while(表达式)
{
    语句
    while(表达式)
    {
        语句
    }
}
```

(2) do...while 结构中嵌套 do...while 结构。

```
do
{
    语句
    do
    {
        语句
    }
    while(表达式);
}
while(表达式);
```

(3) for 结构中嵌套 for 结构。

```
for (表达式;表达式;表达式)
{
    语句
    for(表达式;表达式;表达式)
    {
        语句
    }
}
```

(4) do...while 结构中嵌套 while 结构。

```
do
{
    语句
    while(表达式);
    {
        语句
    }
}
```



Note



```
}
while(表达式);
```

(5) do-while 结构中嵌套 for 结构。

```
do
{
    语句
    for(表达式;表达式;表达式)
    {
        语句
    }
}
while(表达式);
```



Note

以上是关于一些嵌套的结构方式。当然，还有不同结构的循环嵌套，在此就不对每一项都进行列举，只要将每种循环结构的方式把握好，就可以正确写出循环嵌套。

专家点评

循环语句之间是可以进行嵌套使用的，嵌套使用之前必须对每一种循环语句的结构以及流程都非常了解，这样才能进行有效地循环嵌套。

问题 121 如何正确使用循环嵌套？

问题阐述

循环嵌套的应用非常广泛，如无限循环、查找循环、计数循环等，那么如何正确使用它呢？

专家解答

正确的循环嵌套模式是这样的：在一个循环结构当中包含另外一个循环结构，就像玩具套娃一样，一层套一层，其基本形式如图 8.14 所示。

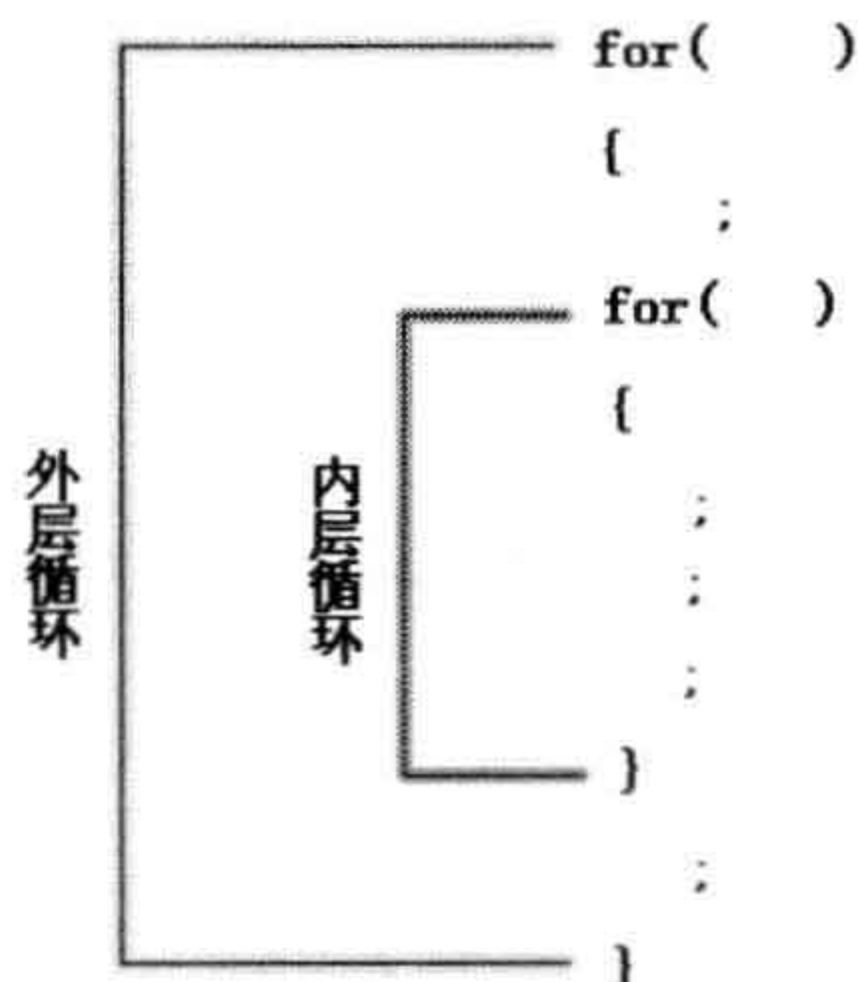


图 8.14 循环嵌套



如果用图 8.15 所示的符号来表示循环，则图 8.16 所示的所有复杂的循环都是允许的。



图 8.15 循环符号

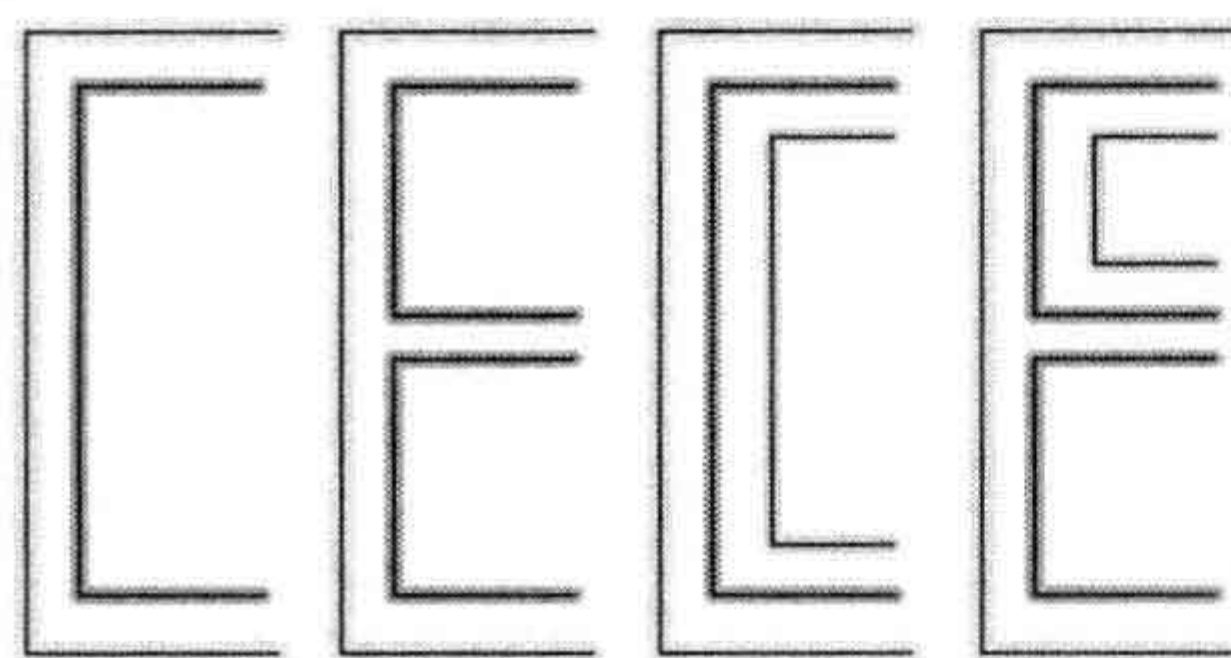


图 8.16 各种复杂的循环嵌套

在使用循环嵌套的时候，需要注意的是，循环不可以有交叉的情况，即两个循环不可能有交叉产生，像图 8.17 所示的循环交叉就是不允许的。

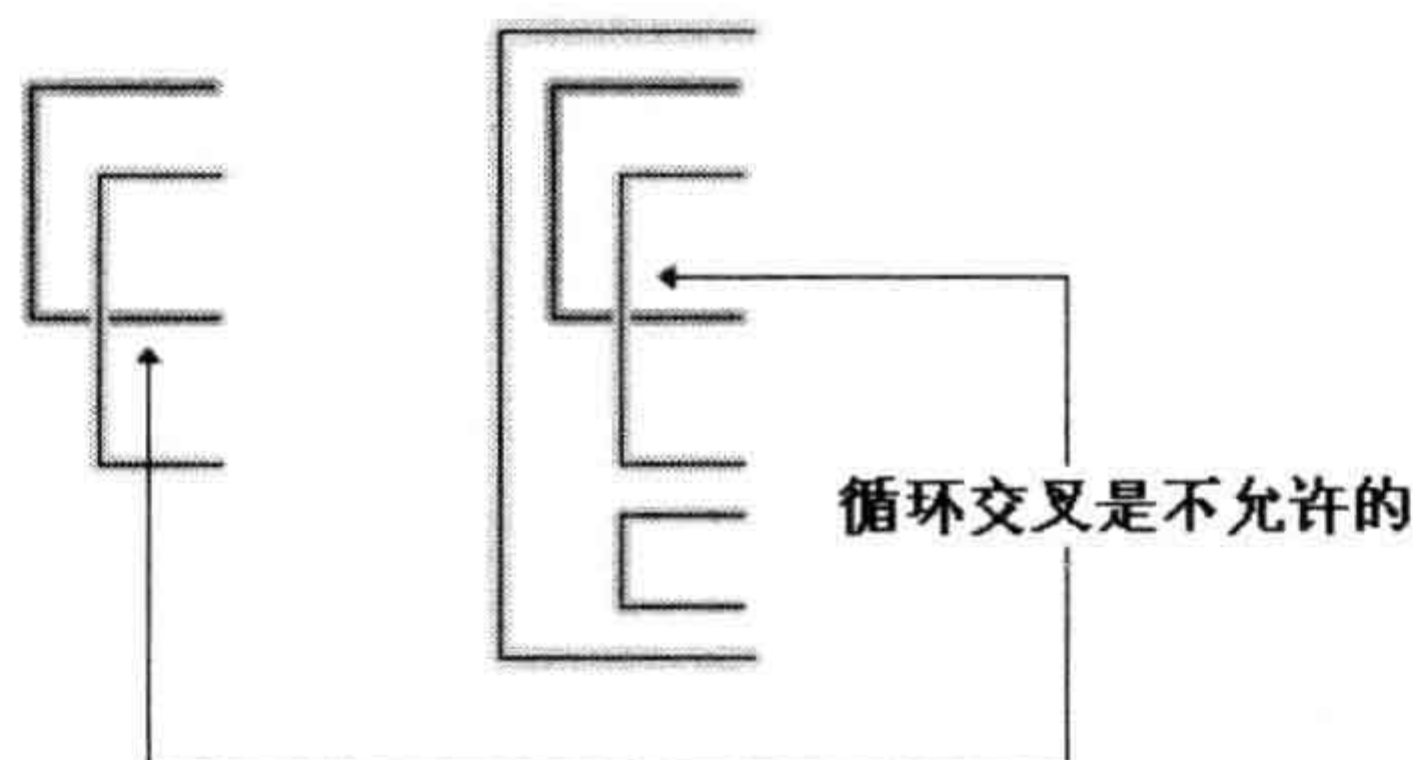


图 8.17 循环交叉是不允许的

例如，下面的这种嵌套形式就是不允许的。

```
do
{
    语句
    for(表达式;表达式;表达式)
    {
        语句
    }while(表达式);
}
```

专家点评

使用循环嵌套要注意层与层之间必须清晰完整，就像跑道一样，有内圈和外圈。还需要注意的就是内层循环和外层循环的控制变量不要重名，否则会造成混乱。

问题 122 死循环是怎样产生的？

问题阐述

死循环是指程序无法退出或者无法进入下一次循环。那么，什么情况会产生死循环呢？



专家解答

1. 问题的产生

C 语言中常用 3 种循环语句, 这些循环语句各有特点, while 和 do...while 经常用在循环次数不确定的场合; for 被经常用来遍历数组和集合。在使用这些循环语句过程中, 若其循环结束条件处理不当, 则很可能产生死循环。下面看一个例子, 代码如下。



Note

```
#include <stdio.h>
void main()
{
    unsigned int i;                /*定义无符号整型变量*/
    for (i=10;i>=0;i--)
    {
        printf ("%d ",i);          /*输出*/
    }
    printf("\n");                  /*输出回车*/
}
```

程序本意是将每次得到的计数变量的值输出出来, 结果由于疏忽导致程序死循环, 程序运行结果如图 8.18 所示。

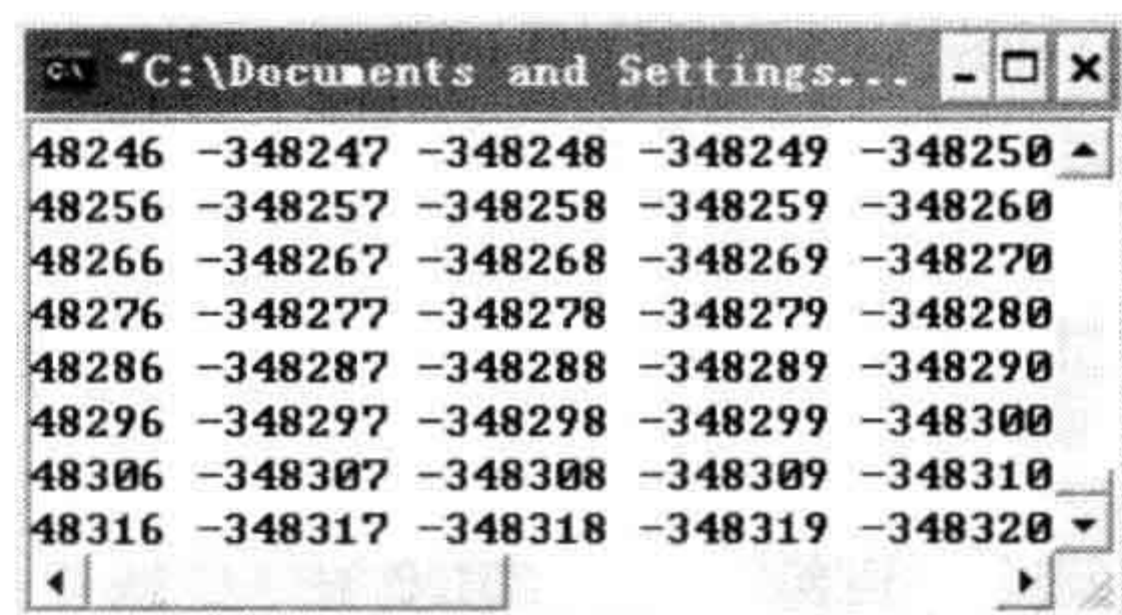


图 8.18 产生的死循环

2. 问题分析与解决方法

for 语句的循环结束条件是当条件表达式的值为 false 时。上面实例中的计数变量 i 被定义为 unsigned 类型, 这样 $i \geq 0$ 就永远成立, 所以程序进入了死循环。

下面是另外两种常用的循环语句的判断条件:

- (1) while 语句的循环结束条件是当布尔表达式的值为 false 时。
- (2) do...while 语句的循环结束条件是当布尔表达式的值为 false 时。

注意:

使用 do...while 语句时要注意循环次数的判断, 因为 do...while 语句是先执行循环体后进行判断。

解决本例的办法就是将计数变量 i 定义为 int 型, 这样就不会出错了。程序的正确运行结果如图 8.19 所示。

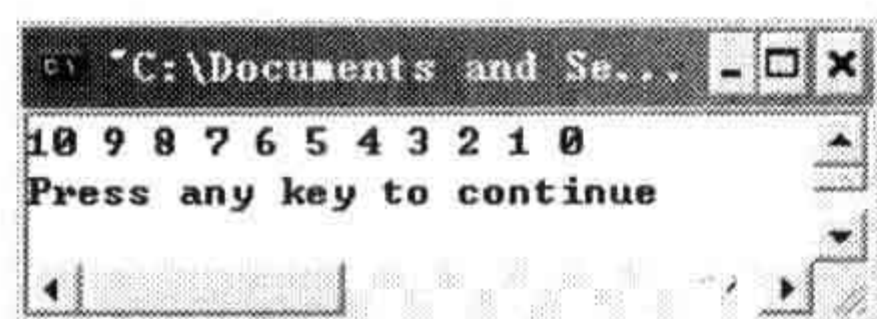


图 8.19 修改后的结果



Note

说明:

在实际的应用程序开发中,产生死循环的情况比较复杂,并不完全像本实例中的情况,但归根结底都是由循环结束条件处理不当造成的,下面将对这些方法进行详细地分析。

专家点评

在循环程序中应避免出现死循环,即应保证循环变量的值在运行过程中可以得到修改,并使循环条件逐步变为假,从而结束循环。

问题 123 怎样提高循环语句的效率?

问题阐述

在 C 语言程序中,常常使用循环结构来解决特定的问题。那么在设计程序时,怎样才能提高循环语句的效率呢?

专家解答

C 循环语句中,for 语句使用频率最高,while 语句其次,do 语句很少用。提高循环体效率的基本办法是降低循环体的复杂性。下面说明提高循环效率的几种情况。

1. 直接提高循环语句的效率

(1) 在多重循环中,如果有可能,应当将最长的循环放在最内层,最短的循环放在最外层,以减少 CPU 切换循环层的次数。例如下面两个循环语句,循环语句 2 比循环语句 1 的效率高。

语句 1: 长的循环在外层

```
for (i=0; i<100; i++)
{
    for (j=0; j<5; j++)
    {
        sum = sum + a[i][j];
    }
}
```

语句 2: 长的循环在内层

```
for (j=0; j<5; j++)
{
```




```
for (i=0; i<100; i++)
{
    sum = sum + a[i][j];
}
```

(2) 如果循环体内存在逻辑判断, 并且循环次数很大, 宜将逻辑判断移到循环体的外面。例如下面两个循环语句:

语句 1: 效率低但程序简洁

```
for (i=0; i<N; i++)
{
    if (condition)
        DoSomething();
    else
        DoOtherthing();
}
```

语句 2: 效率高但程序不简洁

```
if(condition)
{
    for (i=0; i<N; i++)
        DoSomething();
}
else
{
    for (i=0; i<N; i++)
        DoOtherthing();
}
```

语句 1 比语句 2 多执行了 $N-1$ 次逻辑判断。并且由于前者总要进行逻辑判断, 打断了循环的模式, 使得编译器不能对循环进行优化处理, 降低了效率。

如果 N 非常大, 最好采用示例语句 2 的写法, 可以提高效率。如果 N 非常小, 两者效率差别并不明显, 采用语句 1 的写法比较好, 因为程序更加简洁。

2. 间接提高循环语句的效率

(1) 对于简单的 if...else 语句, 建议使用 c 语言的三目运算符 “?:” 代替, 以提高程序的执行效率。如:

```
if(i<20)
{
    a=1;
}
else
{
    a=6;
}
```



Note



可以写成:

```
a=(i<20)?1:6;
```

(2) 在程序中, 不允许出现如下风格的语句。

```
if(i<20)
    return a;
return b;
```

应写成:

```
if(i<20)
{
    return a;
}
else
{
    return b;
}
```

(3) 对于 switch 语句, 在每个 case 语句的后面不要忘记加 break, 除非是想使得某几个分支重叠。

(4) switch 语句一定要有 default, 即使它不做什么。

专家点评

从上面的学习中可以知道, 提高循环语句的效率有时是按照特定的方法降低代码的复杂性, 有时却是视情况而定的。总的来说, 提高循环体的效率应当在书写程序的循环语句时保持良好的风格。

问题 124 continue 语句的基本作用是什么?

问题阐述

在某些情况下, 程序需要用到 continue 语句进行跳转, 那么 continue 语句的具体作用是什么呢?

专家解答

continue 的作用是结束本次循环, 即跳过循环体中下面尚未执行的部分, 接着执行下一次的循环操作。

例如, 使用 continue 结束本次的循环操作, 代码如下:

```
#include<stdio.h>
int main()
```




```
{
    int iCount;                                /*循环控制变量*/
    for(iCount=0;iCount<10;iCount++)          /*执行 10 次循环*/
    {
        if(iCount==5)                          /*判断条件, 如果 iCount 等于 5 跳出*/
        {
            printf("Continue here\n");          /*跳出本次循环*/
            continue;
        }
        printf("the counter is:%d\n",iCount);    /*输出循环的次数*/
    }
    return 0;
}
```



Note

程序运行结果如图 8.20 所示。

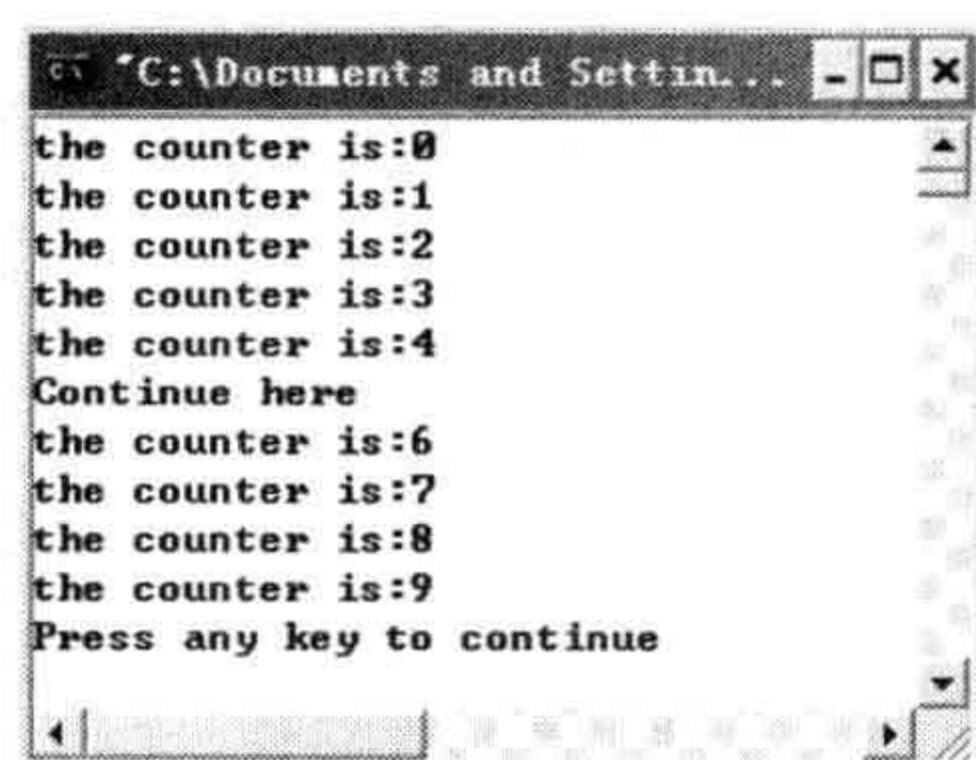


图 8.20 使用 continue 结束本次的循环操作

通过程序的显示结果, 可以看到在 iCount 等于 5 时通过调用 continue 语句, 使得当次的循环结束。但是循环本身还没有完, 所以会继续执行。

专家点评

continue 语句结束的是本次循环, 而不是终止整个循环的执行。与 break 语句不同的是, break 语句结束的是整个循环过程, 将不再判断执行循环的条件是否成立。

问题 125 break 语句的基本作用是什么?

问题阐述

break 语句用于在程序中跳转, 那么它具体是怎样跳转的呢? continue 语句也用于跳转, 它们有什么区别呢?

专家解答

1. break 语句

break 语句终止并跳出循环, 继续执行后面的代码。break 语句的一般形式为:



break;

break 语句不能用于循环语句和 switch 语句之外的任何其他语句中。例如，在循环使用 break 语句。



Note

```
while(1)
{
    printf("Break");
    break;
}
```

在代码中，虽然 while 语句是一个条件永远为真的循环，但是在其中使用 break 语句使得程序流程跳出循环。

这个 break 语句和 switch...case 分支结构中的 break 语句作用是不同的。当循环体中嵌套 switch 语句，并且 break 出现在 switch 语句中，执行到 break 时，则只跳出 switch 语句块，并非跳出循环。

下面通过具体实例来介绍如何使用 break 语句跳出循环。

使用 for 语句执行循环输出 10 次的操作，在循环体中判断输出的次数。如果当循环变量为 5 次时，使用 break 语句跳出循环，终止循环输出操作，代码如下。

```
#include<stdio.h>
int main()
{
    int iCount;                /*循环控制变量*/
    for(iCount=0;iCount<10;iCount++) /*执行 10 次循环*/
    {
        if(iCount==5)          /*判断条件，如果 iCount 等于 5 跳出*/
        {
            printf("Break here\n");
            break;              /*跳出循环*/
        }
        printf("the counter is:%d\n",iCount); /*输出循环的次数*/
    }
    return 0;
}
```

变量 iCount 在 for 语句中被赋值为 0，因为 iCount<10，则循环执行 10 次。在循环语句中使用 if 语句判断当前 iCount 的值。当 iCount 值为 5 时，if 判断为真，使用 break 语句跳出循环。

运行程序，显示结果如图 8.21 所示。

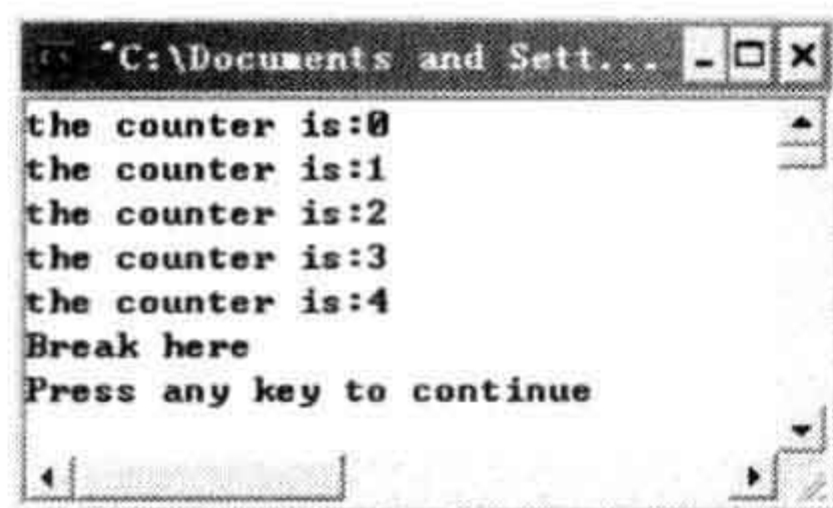


图 8.21 使用 break 语句跳出循环



2. break 语句与 continue 语句的区别

在 switch 语句中, break 语句可用在 case 分支的后面, 用于执行完某个分支后跳出 switch 语句, 而 continue 语句却不能用在 switch 语句中。

在循环语句中, break 语句用于结束循环。continue 用于跳过本次循环体中余下尚未执行的语句, 然后进行下一次的循环条件判定。continue 语句只是结束本次循环却并未跳出循环, 终止程序。



Note

注意:

当有循环嵌套时, break 语句只跳出本层循环, 并不能结束整个多层循环。

专家点评

在遇到不顾表达式检验的结果而强行终止循环的情况时, 可以使用 break 语句结束循环。但 break 只能跳出本层循环, 若想在多层循环跳出可用 goto 语句。

问题 126 goto 语句的基本格式是什么? 如何使用?

问题阐述

goto 语句为无条件转向语句, 它可以使程序立即跳转到函数内部的任意一条可执行语句, 这样使用起来比较灵活。那么, 该语句的基本格式是什么? 又该如何使用呢?

专家解答

1. goto 语句的基本格式

goto 关键字后面带一个语句标号, 该语句标号是同一个函数内某条语句的标号。标号可以出现在任何可执行语句的前面, 并且以一个冒号“:”作为后缀。

通常的情况下, goto 与条件语句配合使用, 可以用来实现条件转移, 构成循环或者跳出循环体等功能。一般形式为:

```
goto 语句标号;
```

在 switch 结构中, 每个 case 关键字及后面的常量都是一个标号。

语句标号用标识符表示, 要遵循变量名的命名规则, 即由字母、数字和下划线组成, 并且第一个字符不能是数字。如:

```
goto 25;
```

这个语句就是不合法的。

2. goto 语句的使用

goto 后的语句标号就是要跳转的目标, 当然这个语句标号要在程序的其他地方给出, 但是其语句标号要在函数内部。例如:



```
goto Show;
printf("the message before ShowMessage");
Show:
printf("ShowMessage");
```



Note

上面代码中，goto 后的 Show 为跳转的语句标号，而其后“Show:”代码表示 goto 语句要跳转的位置。这样，在上面的语句中，第一个 printf() 函数不会执行，而会执行第二个 printf() 函数。

goto 跳转语句跳转的方向可以向前，也可以向后；可以跳出一个循环，也可以跳入一个循环。

下面通过一个例子介绍使用 goto 语句如何从循环内部跳出。

要求程序在执行循环操作的过程中，当用户输入退出指令后，程序跳转到循环外部执行程序退出前的显示操作，程序代码如下：

```
#include<stdio.h>
int main()
{
    int iStep;                                /*定义变量，表示外部循环步骤*/
    int iSelect;                              /*保存用户的输入选项*/
    for(iStep=1;iStep<10;iStep++)            /*外部步骤循环*/
    {
        printf("The Step is:%d\n",iStep);    /*将其循环的步骤号显示*/
        do                                   /*使用 do...while 语句进行循环*/
        {
            printf("enter a number to select\n"); /*输出提示信息*/
            printf("(0 is quit,99 for the next step)\n");
            scanf("%d",&iSelect);              /*用户输入选择*/
            if(iSelect==0)                     /*判断是否输入的是 0*/
            {
                goto exit;                    /*执行 goto 跳转语句*/
            }
        }
        while(iSelect!=99);                  /*进行判断用户输入*/
    }
    exit:                                    /*跳转语句执行位置*/
    printf("Exit the program!\n");            /*显示程序结束信息*/
    return 0;
}
```

(1) 程序运行时，for 循环控制程序步骤，程序输出的循环步骤为 1。信息提示输入数字，其中 0 表示退出，99 表示下一个步骤。

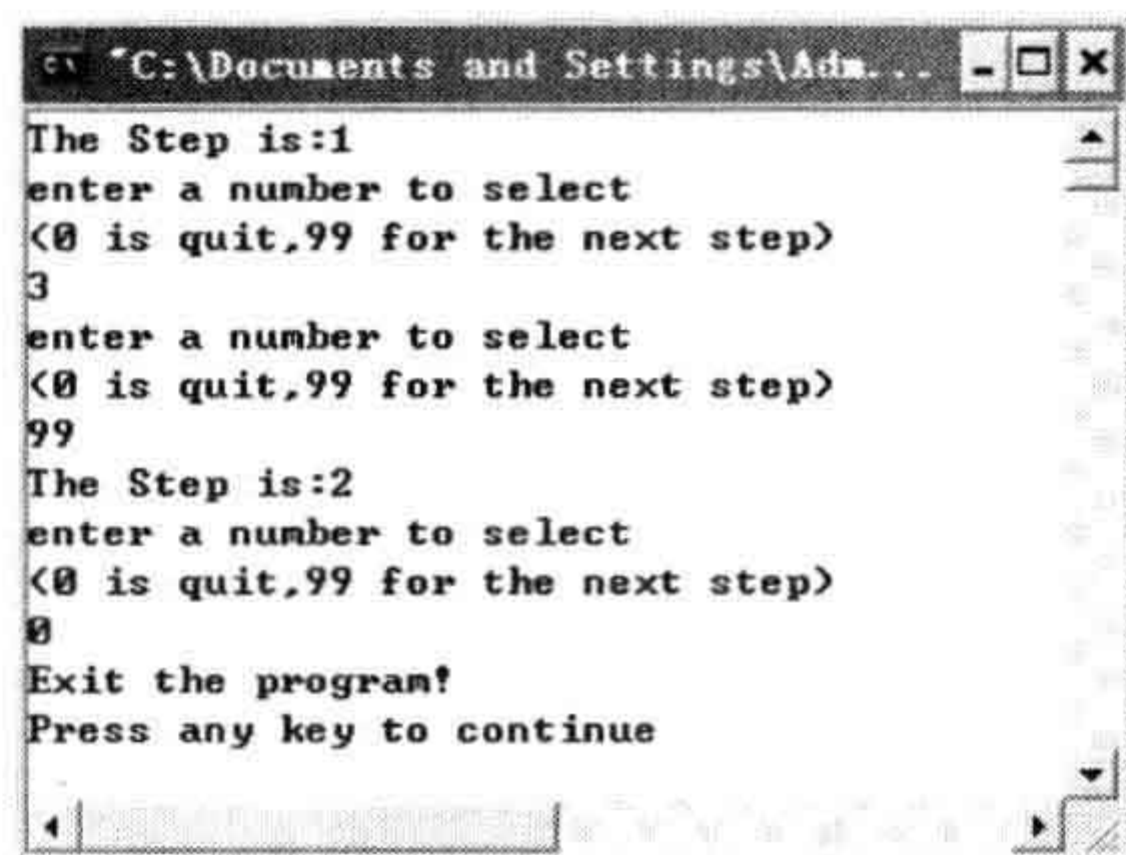
(2) 在 for 循环中使用 do...while 判断用户输入，当条件为假时，循环结束，执行 for 循环的下一步。在程序中假如用户输入数字 3，既不退出也不到下一步骤，程序显示继续输入数字。当输入数字为 99 时，跳转到下一步，显示提示信息“The step is 2”。

(3) 当用户输入的是 0 时，那么通过 if 语句进行判断为真，执行其中的 goto 语句执



行跳转。其中 `exit` 为跳转的语句标号。循环的外部使用 “`exit:`” 表示 `goto` 跳转的位置，然后执行 `printf` 语句，输出一段信息，表示程序结束。

程序运行结果如图 8.22 所示。



```
C:\Documents and Settings\Ada...
The Step is:1
enter a number to select
<0 is quit,99 for the next step>
3
enter a number to select
<0 is quit,99 for the next step>
99
The Step is:2
enter a number to select
<0 is quit,99 for the next step>
0
Exit the program!
Press any key to continue
```

图 8.22 使用 `goto` 语句从循环内部跳出



Note

专家点评

`goto` 语句虽然能够按照自己的意愿改变程序的运行方向，但是这样会使程序流程无规律，可读性较差，所以一般不使用 `goto`，而且它的功能用 `break` 和 `continue` 都能实现，可以用其代替 `goto`。

说明：

除非能较大程度地提高程序的效率，否则不用 `goto` 语句。

问题 127 goto 语句的缺陷是什么？

问题阐述

`goto` 语句可以灵活跳转，适用于多层循环的跳转，随意性强，如果不加限制，会出现什么现象、导致什么结果呢？

专家解答

1. `goto` 的局限性

(1) 使用 `goto` 语句只能将 `goto` 跳转到同一函数内，而不能从一个函数里跳转到另外一个函数里。

(2) 使用 `goto` 语句在同一函数内进行 `goto` 跳转时，`goto` 的起点应是函数内一段小功能的结束处，`goto` 跳转的目的 `label` 处应是函数内另外一段小功能的开始处。

(3) `goto` 不能从一段复杂的执行状态中的位置跳转到另外一个位置，比如：不可以从多重嵌套的循环判断中跳出去。

(4) `goto` 语句有一个特别明显的用途，那就是能从多层循环体中直接跳出，用不着写很多次的 `break` 语句，例如：



Note

```

{...
  {...
    {...
      goto SS;
    }
  }
}
SS:
...
```

goto 其实是很不负责任的, 就像楼房着火了, 来不及从楼梯一阶一阶走下去, 不管楼里面的火势就直接从窗户跳出去一样。所以建议少用、慎用 goto 语句。

2. goto 的缺点

goto 语句的随意性较大, 如果不加以限制, 就会破坏结构化设计风格, 会导致代码晦涩难懂, 降低可读性。例如, 下面的代码:

```

S:
if()
    goto T;
O:
if()
    goto S;
else if()
    goto T;
else
    goto O;
T:
```

这段代码可读吗? 可以画出流程图吗? 答案当然是否定的, 跳来跳去的, 很凌乱, 导致一个程序块有多个出口。这还可能导致在某种情况下忘记执行在块中退出时的操作, 如释放指针等。有时还会因为它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句导致错误或留下隐患, 例如:

```

goto state;
char s1, s2;           /*被 goto 跳过*/
int a = 0;              /*被 goto 跳过*/
...
state:
...
```

如果编译器不能发觉此类错误, 每用一次 goto 语句都可能留下隐患。

专家点评

从上面的学习可以知道 goto 的优点以及弊端。从一般的程序流程来说, goto 语句破坏了清晰的程序结构, 大大降低了程序的可读性, 所以不建议使用 goto 语句。



问题 128 如何选择循环语句?

问题阐述

循环语句有三种, 分别是 `while` 循环、`do...while` 循环和 `for` 循环。对于不同的程序, 应该使用不同的循环结构。那么, 应该如何选择呢?

专家解答

1. 三种循环的比较

三种循环都可以用来处理同一问题。一般情况下, 它们可以相互代替。下面是这三种循环语句的比较:

`while` 和 `do...while` 循环, 只在 `while` 后面指定循环条件, 在循环体中应包含使循环趋于结束的语句 (如 `i++`, 或者 `i=i+1` 等)。`for` 语句中的第 3 个表达式中包含使循环趋于结束的操作, 设置可以将循环体中的操作全部放在表达式 3 中。因此 `for` 语句的功能更强, 凡用 `while` 循环能完成的, 用 `for` 循环都能实现。

用 `while` 和 `do...while` 循环时, 循环变量初始化的操作应在 `while` 和 `do...while` 语句之前完成。而 `for` 语句可以在表达式 1 中实现循环变量的初始化。

`while` 循环、`do...while` 循环和 `for` 循环, 都可以用 `break` 语句跳出循环, 用 `continue` 语句结束本次循环。

2. 三种循环的使用原则

`for` 语句是 C 语言中使用最灵活的循环语句, 它可以用于循环次数已知的情况, 还能用于循环次数不确定的情况, 但要给出循环结束条件。

`while` 语句是一种先判断后执行的语句, 如果开始不能满足条件, 则可以一次都不执行循环体。

`do...while` 语句用法和 `while` 语句相似, 也是要求先给出循环条件, 经过判断后, 根据循环条件是否满足, 来确定是否执行循环体。

专家点评

使用循环语句可以将复杂的、不易理解的解题过程转换为易于理解的多次重复的操作。三种循环虽有各自的特点, 但是可以通用。

问题 129 如何判定循环结束和提前结束?

问题阐述

在程序中, 通常会用到循环结构反复地执行某一语句, 但是不能无限地执行下去, 否



则会出现死循环，程序无法终止。那么，如何判定循环结束和提前结束呢？

专家解答

1. 问题分析

一般情况下，判断循环结束可以由循环条件进行判断，循环语句都有判断循环结束的表达式，如 while 语句中 while(x>10 && x<50) 中的条件表达式 x>10 && x<50；再如，for(i=1;i<10;i++) 中条件表达式 i<10，当不满足条件表达式时就会结束循环，或者使用 break 语句，当程序执行到 break 语句时，跳出循环。例如：

```
#include<stdio.h>
int main()
{
    int a=0,n;
    printf("enter n:\n");
    scanf("%d", &n);
    while(1)
    {
        a++;
        n--;
        printf("%d\n",a*a);
        if(n==0)
            break;
    }
    return 0;
}
```

输入一个 n 值，当 n 值自减到 0 或者 n 的值为 0 时，则会执行 break 语句，跳出循环。

2. 问题解答

判断循环是正常结束还是提前结束，通常是在循环语句的后面再用一个或多个判断语句，检查循环变量的取值。如果循环条件由多个循环变量构成，则可以对各个变量分别进行判断。

下面通过一个例子来说明循环判断结束的条件，代码如下。

```
void main()
{
    int i,a;                                /*定义变量 i, j 为基本整型*/
    printf("please input:\n");
    scanf("%d",&a);
    for(i=2;i<a;i++)
        if(a%i==0)
            break;
    if(i==a)
        printf("素数");
    else
```





```
printf("不是素数");
```

```
}
```

上面的代码用于判断一个数是否为素数。程序中的循环语句正常结束是当 i 大于 a 的时候，但是当 a 不能被 i 整除时，循环就会提前结束，故在循环语句后面添加一条判断语句，以提前结束循环。



Note

专家点评

在程序中，一个完整的循环包括三部分，即循环的入口、循环体和循环的出口。循环若没有结束的条件便成为无限循环，即死循环，程序无法终止。除非某些特定环境需要用到无限循环，否则无限循环没有实际意义。

问题 130 如何避免循环中的初值错误问题？

问题阐述

在一些进行累加、累乘、累减的计算中，要对累积的变量赋初值，然后进行循环，不断地累积计算。但是当忘记为累积变量赋初值，或者赋初值的位置不当时，便会出现累积错误，应如何避免这种错误呢？

专家解答

首先分析一个例子，1~100 之间的整数的累计和，代码如下：

```
#include<stdio.h>
int main()
{
    int i;
    int iSum=0;
    i=1;                                     /*设定循环控制变量的初值*/
    while(i<=100)                           /*while 语句*/
    {
        iSum=iSum+i;                        /*累加计算*/
        i++;                               /*循环变量自增*/
    }
    printf("1 到 100 数字间的累加计算:%d\n",iSum); /*输出计算结果*/
    return 0;
}
```

在程序代码中， $iSum$ 表示累加计算的结果， i 表示 1 到 100 间的数字。为 $iSum$ 赋值为 0， i 赋值为 1。使用 `while` 语句判断 i 是否小于等于 100，如果条件为真，则执行跟着的语句块中的内容；如果条件为假，则跳过语句块执行后面的内容。在语句块中，总和 $iSum$



等于之前的计算的总和加上现在 i 表示的数字，完成累加操作。程序运行结果如图 8.23 所示。

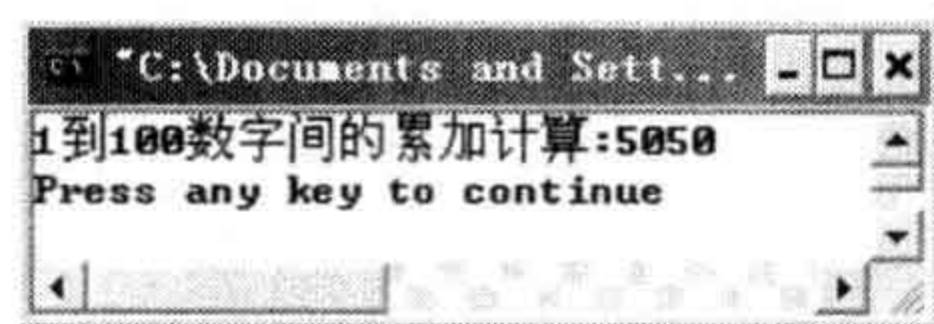


图 8.23 累加计算

假设将累加变量的赋初值语句：

```
int iSum=0;
```

换成：

```
int iSum;
```

会出现什么现象呢？运行程序，改变程序后的结果如图 8.24 所示。

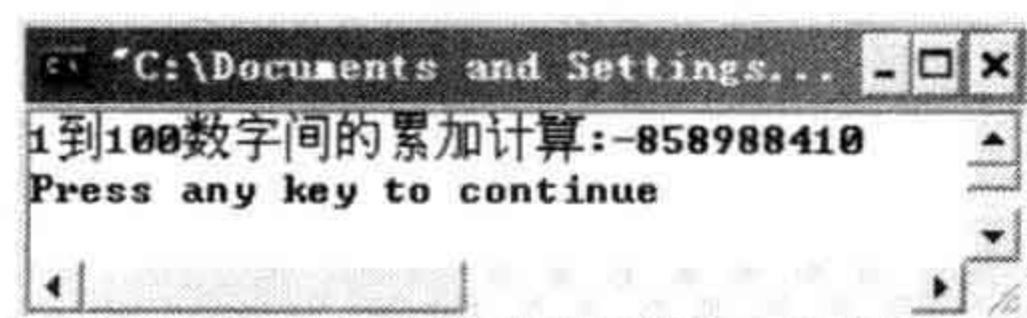


图 8.24 改变代码后

通过结果图可以看到，这不是正确的结果，由于 $iSum$ 的值未赋值，所以它的值不确定，导致最后的结果也不正确。

解决这一问题的办法就是将累加变量赋值，假设将代码改为（黑体部分为更改的代码）：

```
#include<stdio.h>
int main()
{
    int i;
    int iSum;
    i=1;
    while(i<=100)
    {
        iSum=0;
        iSum=iSum+i;
        i++;
    }
    printf("1 到 100 数字间的累加计算:%d\n",iSum);
    return 0;
}
```

/*设定循环控制变量的初值*/
/*while 语句*/
/*给累加变量赋值*/
/*累加计算*/
/*循环变量自增*/
/*输出计算结果*/

程序运行结果如图 8.25 所示。

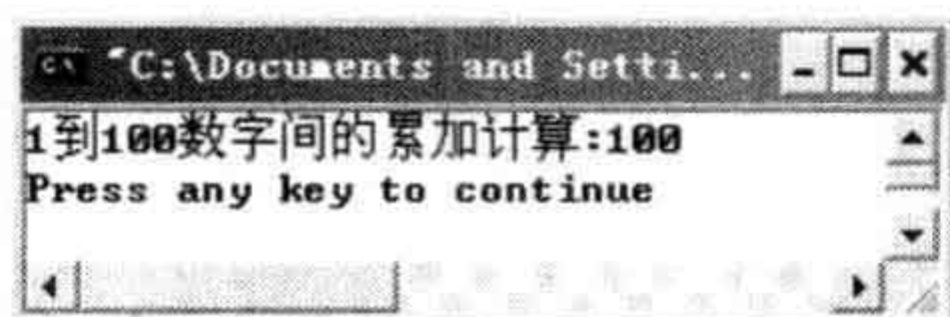


图 8.25 改变代码后的结果





很明显,改变后的程序不对,为什么会这样呢?这是由于将累加变量的赋值语句放在了循环体中, iSum 实际并没有累加,当第二次执行循环时, iSum 的值又变为 0,所以加的是 i 最后一次的值,最后才会出现结果为 100 的情况。

专家点评

从上面的学习可以了解到,在循环结构中,若累积变量未赋初值或者位置不当,将会导致累积错误,不能产生预期的效果。为了避免这种情况的发生,一定要为累积变量赋初值,且在循环体内不断改变它的值。



Note

第9章

数组

- ▶▶ 什么是数组？其存储有何特点？
- ▶▶ 数组的维数该如何理解？
- ▶▶ 一维数组是怎样定义的？
- ▶▶ 如何引用一维数组元素？
- ▶▶ 如何初始化一维数组？
- ▶▶ 如何设计数组的排序算法？
- ▶▶ 如何定义二维数组？
- ▶▶ 如何引用二维数组元素？
- ▶▶ 如何初始化二维数组？
- ▶▶ 如何定义字符数组？
- ▶▶ 如何初始化字符数组？
- ▶▶ 如何引用字符数组？
- ▶▶ 如何进行字符数组的复制？
- ▶▶ 如何进行字符数组的连接？
- ▶▶ 如何进行字符串的比较？
- ▶▶ 如何测定字符串的长度？
- ▶▶ 如何进行字符串大小写的相互转换？
- ▶▶ 如何计算字符串中有多少个单词？
- ▶▶ `gets()`和 `scanf()`在输入字符串时有何区别？
- ▶▶ `puts()`和 `printf()`在输出字符串时有何区别？
- ▶▶ 数组与指针的区别是什么？
- ▶▶ 为什么作为函数形参的数组和指针可以互换？
- ▶▶ 为什么数组名作参数传递给子函数时，子函数可以改变主函数中数组的值？
- ▶▶ C语言中有动态数组吗？
- ▶▶ 如何实现动态二维数组？
- ▶▶ `strcpy()`函数可以复制字符串的一部分吗？
- ▶▶ 字符串和字符数组有什么区别？
- ▶▶ ‘\0’和“\0”有什么区别？
- ▶▶ 字符数组占用内存怎样算？
- ▶▶ 用字符数组和指针两种方式定义的字符串有什么不同？



问题 131 什么是数组？其存储有何特点？

问题阐述

什么是数组，数组和数列有什么关系？计算机中是怎样表示数组的？

专家解答

在使用计算机对数据进行处理时，经常会遇到如下的情况：具有相同数据类型的数据量很大，如全班同学的考试成绩、公司中全部员工记录等，这时就要用到数组。

数组可以理解为类型相同、数目固定的有限个变量的集合。在 C 语言中，一个数组有一个统一的数组名，数组中的每一个元素用一个确定的下标来标识。

数组元素是组成数组的基本单元，同一数组中的数据元素必须具有相同的数据类型，数组元素由“数组名”和“下标”唯一确定。

数组是有序数据的集合，有序是指系统在存放的时候会为数据元素分配一段连续的存储空间，数据元素在这段空间内按照先后顺序进行存放。

数组和数列是完全不同的概念，数学中的数列是一组有规律的数，而计算机中的数组只是很多数的一种表示方式，不要求有规律。

专家点评

数组是程序设计语言中处理大量数据时使用的一种数据形式。数组中的数据要求类型一致、容量有限。它的作用是保存大量数据。数学中的数列是研究数的变化规律的。

问题 132 数组的维数该如何理解？

问题阐述

什么叫做维，维是不是数组中数的个数呢？

专家解答

维数是数组元素的下标个数。使用数组的时候，如果只有一个下标，则称为一维数组，一维数组一般表示一种线性数据的组合。二维数组则是有两个下标，可以将其看做是平面数据的组合。三维数组有三个下标，可以看做是立方体。

对现实问题，问题整体可以由几个不同方面去描述，以便区分不同个体。对一组考试成绩，如果只由姓名来区分，就是一维数组；如果用姓名、科目来区分就是二维数组；用姓名、科目、考试时间来区分，就是三维数组.....C 语言对数组下标的数量没有限制。



Note



专家点评

一维、二维、三维可以表示为几何中的直线、平面、立体。四维及以上的数组没有办法用对应的几何方式描述。



Note

问题 133 一维数组是怎样定义的？

问题阐述

此处所说的定义，不是指一维数组概念（什么是一维数组），而是指在 C 语言中怎样定义一个数据是一维数组，定义格式什么样？

专家解答

C 语言使用变量时有一条规定：“先定义，后使用。”定义数组和定义变量一样，定义数组的目的是为某组数据申请内存空间并命名这块内存空间，以便在后面的程序中使用。

在定义数组时，要明确向编译器表明两点。

- (1) 数组元素的类型。
- (2) 数组元素的总个数。

一维数组定义的一般形式如下：

类型说明符 数组标识符[常量表达式];

- ☑ 类型说明符：表示数组中的所有元素类型，可以是任意一种简单类型、构造类型或指针。
- ☑ 数组标识符：表示这个数组变量的名称，有时也叫做数组名，其命名规则与变量名一致（由字母、数字、下划线组成，但是必须是以字母或者下划线开头）。
- ☑ 中括号（方括号）：这对中括号是数组标志，同时也是数组的重要组成部分，不能缺少，也不可以转换成其他符号。
- ☑ 常量表达式：中括号内的表达式必须为常量和符号常量，不能为变量。常量表达式定义了数组中存放的数据元素的个数，即数组长度。例如 `a[5]`，5 表示数组中有 5 个元素，下标从 0 开始，到 4 结束。

例如定义一个数组。

```
int a[5];
```

代码中的 `int` 为数组元素的类型，而 `a` 表示的是数组变量名，括号中的 5 表示的是数组中包含的元素个数。在数组 `a[5]` 中只能使用 `a[0]`、`a[1]`、`a[2]`、`a[3]`、`a[4]` 而不能使用 `a[5]`。

也可以用 `#define` 宏定义一个符号常量，这个符号常量可以用来声明数组的元素个数。如下所示。

```
#define SIZE 10
```




```
int a[SIZE];
```

上述代码首先通过宏定义`#define`定义符号常量`SIZE`为10,再使用符号常量`SIZE`声明数组元素个数,`a[SIZE]`相当于`a[10]`。中括号里面也可以是常量表达式,例如“`a[6+4]`”也等同于`a[10]`。以上两种声明都是具有10个元素的数组。

对于一维数组,可以使用下列公式计算所需要的内存字节数。

总字节数=sizeof(类型)×数组长度。

一维数组的一般形式中,括号内是常量或常量表达式,不允许出现变量,即不能定义动态数组。

例如,下面定义的方法是错误的:

```
int i=9;  
int a[i];
```

专家点评

以下两点必须注意。

- (1) 数组下标从0开始。
- (2) 定义时必须决定数组中数的个数。

问题 134 如何引用一维数组元素?

问题阐述

数组是一组数的集合,数组元素是这组数中的一个个体。怎样引用数组集合中的每一个个体呢?

专家解答

数组定义完成后就要使用该数组,可以通过引用数组元素的方式,使用该数组中的元素。在引用数组的时候,应该注意下标的范围,其范围是从0~(元素个数-1)。在C语言中,数组元素不能够整体被引用,只能逐个引用。

数组元素表示的一般形式如下。

数组标识符[下标]

例如,引用一个数组变量`a`中的第3个变量,可以表示为:

```
a[2];
```

如果数组中有10个数,定义为:

```
int a[10];
```



Note



逐个引用数组元素的方法为：

```
for(i=0;i<10;i++)  
    printf("%d",a[i]);
```

专家点评

由于数组中数据量大，所以对数组的引用都是通过循环语句来完成的。

问题 135 如何初始化一维数组？

问题阐述

初始化数组就是在定义数组变量的同时给其中的数组元素赋值。怎样对一维数组进行初始化呢？

专家解答

一维数组初始化的一般形式如下：

```
类型说明符 数组名[常量表达式]={值 1,值 2,...,值 n};
```

对一维数组的初始化主要有三种形式，下面对这三种形式进行详细讲解。

(1) 在定义数组时，可直接对全部的数组元素赋初值，即全部初始化。

例如：

```
int i,a[6]={1,2,3,4,5,6};
```

该方法是将数组中的元素值一次放在一对花括号中。经过上面的定义和初始化之后，数组中的元素为 $a[0]=1$ ， $a[1]=2$ ， $a[2]=3$ ， $a[3]=4$ ， $a[4]=5$ ， $a[5]=6$ 。

(2) 可以只给一部分元素赋值，当“{}”中值的个数少于元素个数时，只给前面部分元素赋初值，没有赋值的元素默认值为 0。

例如：

```
int a[6]={10,20,30};
```

(3) 在对全部数组元素赋初值时，可以不指定数组长度，即初始化可以决定数组的长度。

例如：

```
int a[]={10,20,30,40,50};
```

专家点评

总之，数组初始化可以一次对所有数据赋值，也可以给部分数据赋值，还可以决定数组元素的个数。



Note



问题 136 如何设计数组的排序算法?

问题阐述

排序是把原本没有顺序的数据重新排列成按大小排列的数据。C语言不能自动对数组进行排序,要自己设计排序算法,实现数组的排序。那么,怎样对数组进行排序呢?



Note

专家解答

数组的排序方法有很多种,这里只讲一种实现简单、广泛使用的算法:交换排序法。例如,有一组数据,用数组表示为:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
3	5	6	2	6	1	6	3

现在要将其按从小到大的顺序进行排列。

交换排序法的基本思路是:

(1) 第一遍。

让 a[1]、a[2]、...、a[7] 每个数均与 a[0] 比较,如果小与 a[0],就让它们交换,结果为:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
2	5	6	3	6	1	6	3

1 2 3

从 a[1] 到 a[3] 与 a[0] 比较, a[1] 和 a[2] 不变, a[3] 与 a[0] 交换,变成 a[0]=2, a[3]=3, 继续

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
1	5	6	3	6	2	6	3

1 2 3 4 5

a[4] 到 a[5] 与 a[0] 比较, a[4] 不变, a[5] 与 a[0] 交换,变成 a[0]=1, a[5]=2, 继续

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
1	5	6	3	6	2	6	3

1 2 3 4 5 6 7

a[6] 到 a[7] 与 a[0] 比较, 不用交换。

这样,经过从头到尾一轮的比较,使 a[0] 变成了数组中最小的数。

用程序描述这一过程就是:

```
for(j=1;j<8;j++)
    if(a[j]<a[0])
        {t=a[0];a[0]=a[j];a[j]=t;}
```

(2) 第二遍。

越过 a[0], 让 a[2] 到 a[7] 每个数与 a[1] 比较,如果小于 a[1],就让这两个数交换。这样



又可以得到第二小的数放在 $a[1]$ 上, 结果为:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$
1	2	6	5	6	3	6	3

用程序描述这一过程就是:

```
for(j=2;j<8;j++)
    if(a[j]<a[1])
        {t=a[1];a[1]=a[j];a[j]=t;}
```

(3) 第三遍。

越过 $a[0]$, $a[1]$, 让 $a[3]$ 到 $a[7]$ 每个数与 $a[2]$ 比较, 如果小于 $a[2]$, 就让这两个数交换。这样又可以得到第三小的数放在 $a[2]$ 上, 用程序描述这一过程就是:

```
for(j=3;j<8;j++)
    if(a[j]<a[2])
        {t=a[2];a[2]=a[j];a[j]=t;}
...
```

(4) 一直到最后剩下一个数为止, 这个数就是最大的了, 整个数组也就完成排序了。

那上面的程序要写多少次呢? 7 次。要是 100 个数排序呢? 要写 99 次, 这是不可容忍的。

如果上面的程序写了 7 次, 我们会发现程序中第 i 次执行时, 其实就是从 $i+1$ 开始到 $n-1$ (n 是数的个数) 为止的所有的数与 $a[i]$ 比较, 也就是可以把 7 次执行再放入一个循环语句中, 写成:

```
for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
        if(a[j]<a[i])
            {t=a[i];a[i]=a[j];a[j]=t;}
```

程序中的 n 是数组中数的个数。

专家点评

排序方法有很多种, 以上讲的是交换排序法。

问题 137 如何定义二维数组?

问题阐述

有一个考试, 共有 10 人参加, 10 个人的考试成绩可以用一维数组表示。如果说还是 10 个人考试, 每个人考 5 科, 这样就有 50 个考试成绩, 这时可以用二维数组来表示。在 C 语言中, 怎样定义二维数组呢?





专家解答

二维数组的定义和一维数组大致相同，只是比一维数组多了一个常量表达式，其一般形式如下：

```
数据类型说明符 数组名[常量表达式 1][常量表达式 2];
```

“类型说明符”用来说明数组中元素的数据类型。“数组名”唯一标识该二维数组。第一个方括号“[]”中的常量表达式 1 标识第一维下标的长度，第二方括号“[]”中常量表达式 2 标识第二维下标的长度。

“常量表达式 1”被称为行下标，“常量表达式 2”被称为列下标。如果有二维数组 $a[n][m]$ ，则二维数组的下标取值范围如下：

- (1) 行下标的取值范围是 $0 \sim n-1$ 。
- (2) 列下标的取值范围是 $0 \sim m-1$ 。
- (3) 二维数组的最大下标元素是 $a[n-1][m-1]$ 。

10 个人考试的问题就可以定义为：

```
int a[10][5];
```

这是一个 10 行 5 列的数组，数组名为 a ，其下标变量的类型为整型。该数组的下标变量共有 10×5 个，即：

```
a[0][0], a[0][1], a[0][2], ..., a[0][9]
a[1][0], a[1][1], a[1][2], ..., a[1][9]
a[2][0], a[2][1], a[2][2], ..., a[2][9]
a[3][0], a[3][1], a[3][2], ..., a[3][9]
a[4][0], a[4][1], a[4][2], ..., a[4][9]
```

在 C 语言中，二维数组是按行排列的，即按行顺次存放，先存放 $a[0]$ 行内所有数组元素，再存放 $a[1]$ 行的所有数组元素。

二维数组可以看做是一维数组的拓展，也可以看成是特殊的一维数组。例如“`int a[10][5];`”，可以理解为一维数组中四个元素 $a[0]$ 、 $a[1]$ 、...、 $a[9]$ ，每个元素又包含五个数组元素，如表 9.1 所示。

表 9.1 二维数组由一维数组组成

二维数组名	一维数组名	数 组 元 素
a	a[0]	a[0][0], a[0][1], a[0][2], a[0][3], a[0][9]
	a[1]	a[1][0], a[1][1], a[1][2], a[1][3], a[1][9]
	a[2]	a[2][0], a[2][1], a[2][2], a[2][3], a[2][9]
	a[3]	a[3][0], a[3][1], a[3][2], a[3][3], a[3][9]
a	a[4]	a[4][0], a[4][1], a[4][2], a[4][3], a[4][9]

这里 $a[0]$ 、 $a[1]$ 、 $a[2]$ 不能当做数组元素使用，只能作为数组名，因为它们已经不是单纯的数了。



Note



专家点评

多维数组的定义与二维数组相似。



Note

问题 138 如何引用二维数组元素？

问题阐述

二维数组中有很多数组元素，怎样使用其中的每个数组元素呢？

专家解答

与一维数组相同，定义了二维数组后就要用它存储数据、管理数据。二维数组的元素也称为双下标变量，二维数组元素的一般形式为：

数组名[下标][下标];

例如，对一个二维数组的元素进行引用：

a[1][2];

这行代码表示的是对 a 数组中第 2 行的第 3 个元素进行引用。

注意：

不管是行下标或者是列下标，其索引都是从 0 开始的。

和一维数组一样这里要注意下标越界的问题，例如：

```
int a[2][4];  
...                               /*对数组元素进行赋值*/  
a[2][4]=9;                         /*错误！*/
```

上面代码这种表示是错误的。因为 a 为 2 行 4 列的数组，它的行下标的最大值为 1，列下标的最大值为 3，所以 a[2][4]超过了数组的范围，下标越界。

定义数组 a[2][4]和引用元素 a[2][4]时是不同的。

- ☑ 在定义数组 a[2][4]时，2 和 4 分别代表的是数组的维数和维数的范围；
 - ☑ 在引用元素 a[2][4]时，2 和 4 是数组元素的下标值，仅仅代表数组中的一个元素。
- 如果数组中有 10 行，每行有 5 个数，定义为：

```
int a[10][5];
```

对所有数据的引用方法为：

```
for(i=0;i<10;i++)  
    for(j=0;j<5;j++)  
        printf("%d",a[i][j]);
```




专家点评

二维数组的遍历引用都是结合两层的循环嵌套完成的。递推到 n 维数组，可以用 n 个循环嵌套完成。

问题 139 如何初始化二维数组？



Note

问题阐述

二维数组怎样初始化，有几种方法？

专家解答

二维数组和一维数组一样，也可以在声明时对其进行初始化。二维数组的初始化方式比一维数组较为复杂，但都是由一维数组初始化方法衍生而来的。在给二维数组赋初值时，有以下 3 种形式：

(1) 按行连续赋值，即将所有数据写在一个大括号内，按照数组元素排列顺序对元素赋值。如：

```
int a[2][2] = {1,2,3,4};
```

如果花括号内的数据少于数组元素的个数，系统将默认后面没被赋值的元素值为 0，以矩阵形式表示为：

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

(2) 在为所有元素赋初值时，可以省略行下标，但是不能省略列下标。如：

```
int a[][3] = {1,2,3,4,5,6};
```

系统会根据数据的个数进行分配，一共有 6 个数据，而数组每行分为 3 列，当然可以确定数组为 2 行，以矩阵形式表示为：

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

(3) 按行分段赋值，可以分行给数组元素赋值，如：

```
int a[2][3] = {{1,2,3},{4,5,6}};
```

在分行赋值时，可以只对部分元素赋值，如：

```
int a[2][3] = {{1,2},{4,5}};
```

在上行代码中，各个元素的值如下：

$a[0][0]$ 的值是：1；

$a[0][1]$ 的值是：2；



a[0][2]的值是: 0;
a[1][0]的值是: 4;
a[1][1]的值是: 5;
a[1][2]的值是: 0;
以矩阵形式表示为:

$$\begin{pmatrix} 1 & 2 & 0 \\ 4 & 5 & 0 \end{pmatrix}$$



Note

专家点评

对二维数组的初始化方式进行扩展, 很自然地就能得到多维数组的初始化方法。

问题 140 如何定义字符数组?

问题阐述

C 语言中的字符型只能表示一个字符, 当表示人的姓名、住址这种多个字母才能表示的数据时, 需要使用字符数组。在 C 语言中, 怎样定义一个变量是字符数组呢?

专家解答

字符数组的定义与其他数据类型的数组定义类似, 一般形式如下:

```
char 数组标识符[常量表达式];
```

因为要定义的是字符型数据, 所以在数组标识符前所用的类型是 `char`, 后面括号中表示的是数组元素的数量。

例如, 定义一个字符数组 `a`:

```
char a[5];
```

其中的 `a` 是数组的标识符, 而括号中的 5 则表示数组中包含 5 个字符型的变量元素。

专家点评

字符数组的定义形式与数值型数组没有什么区别。

问题 141 如何初始化字符数组?

问题阐述

字符数组如何初始化, 与数值型数组有什么不同?



专家解答

字符数组的初始化操作有以下几种方法:

(1) 逐个字符赋给数组中各元素。

这是最容易理解的初始化字符数组的方式。例如, 初始化一个字符数组。

```
char ca[12]={ 'H','E','L','L','O',',', 'M','T','N','G','R','T'};
```

定义包含 12 个元素的字符数组, 在初始化的花括号中, 每一个字符对应赋值一个数组元素。

(2) 如果在定义字符数组时进行初始化, 可以省略数组长度。

如果初值个数与预定的数组长度相同, 在定义时可以省略数组长度, 系统会自动根据初值个数确定数组长度。例如, 上面初始化字符数组的代码可以写成:

```
char ca[]={ 'H','E','L','L','O',',', 'M','T','N','G','R','T'};
```

中间的那个字符是逗号, 在代码中可以看到定义的 `ca[]` 中没有给出数组的大小, 但是根据初值的个数会确定数组的长度为 12。

(3) 利用字符串给字符数组赋初值。

通常用一个字符数组来存放一个字符串。例如, 用字符串的方式对数组作初始化赋值如下。

```
char ca []={"HELLO,MINGRI"};
```

或者将 {} 去掉, 写成:

```
char ca []="HELLO,MINGRI";
```

专家点评

字符数组的初始化用字符串的形式比较简单方便。

问题 142 如何引用字符数组?

问题阐述

字符数组的引用与数值型数组有什么区别, 是否也只能逐个引用?

专家解答

字符数组的引用可以和其他类型数据引用一样, 使用下标的形式。例如, 引用上面定义的数组 `a` 中的元素。

```
a[0]='H';  
a[1]='e';  
a[2]='l';
```



Note



```
a[3]='l';  
a[4]='o';
```

上面的代码依次引用数组中各元素，为其进行赋值。

C 语言函数库支持对字符数组的整体引用，如：

输出：printf(“%s”,a);

输入：scanf(“%s”,a);

赋值：strcpy(a,“hello”);

以上 a 均为数组名，请注意输入和输出语句中都只用数组名，输入不用&a 的形式。

专家点评

字符数组在库函数支持下可以整体引用，这为字符串处理带来了极大的方便。

问题 143 如何进行字符数组的复制？

问题阐述

有两个字符数组 a 和 b，a 的值是 “Good Bye”，b 的值是 “Bye Bye”，现在要把 b 复制到 a 中，使 a 变成 “Bye Bye”，应该怎么做？

专家解答

在字符串操作中，字符串复制是比较常用的操作之一。在字符串处理函数中包含 strcpy() 函数，该函数将复制特定长度的字符串到另一个字符串中。其语法格式如下：

```
strcpy(目的字符数组名, 源字符数组名)
```

strcpy() 函数的功能是把源字符数组中的字符串复制到目的字符数组中，并将字符串结束标志 ‘\0’ 也一同复制。

使用 strcpy() 函数必须包含头文件 string.h。

下面通过实例来介绍一下 strcpy() 函数的使用。

例如，在 main() 函数体中定义两个字符数组，分别用来存储源字符串和目的字符数组，然后获取用户为这两个字符数组赋值的字符串，并分别输出两个字符数组。调用 strcpy() 函数将源字符数组中的字符串赋值给目的字符数组，最后输出目的字符数组。具体代码如下。

```
#include<stdio.h>  
#include<string.h>  
int main()  
{  
    char str1[100],str2[100];  
    printf("输入目的字符串: \n");  
    gets(str1);  
    /*输入目的字符串*/
```




```

printf("输入源字符串: \n");
gets(str2);
printf("输出目的字符串: \n");
puts(str1);
printf("输出源字符串: \n");
puts(str2);
strcpy(str1, str2);
printf("调用 strcpy 函数进行字符串复制: \n");
printf("复制字符串之后的目的字符串: \n");
puts(str1);
return 0;
}

```

/*输入源字符串*/

/*输出目的字符*/

/*输出源字符串*/

/*调用 strcpy()函数实现字符串复制*/

/*输出拷贝后的目的字符串*/

/*程序结束*/

定义两个字符串数组 str1 和 str2, str1 是代表最终要复制出其他数组中字符串的数组,在此称之为目的字符。str2 是被复制的字符数组,在程序中称为源字符串。

程序先让用户输入目的字符串和源字符串,并分别保存到数组 str1 和 str2 中,然后将两个字符串数组输出。接下来,使用 strcpy()函数将 str2 内的字符串复制到 str1 中,最后输出 str1 从 str2 复制的字符串。

运行程序,字符串复制结果如图 9.1 所示。

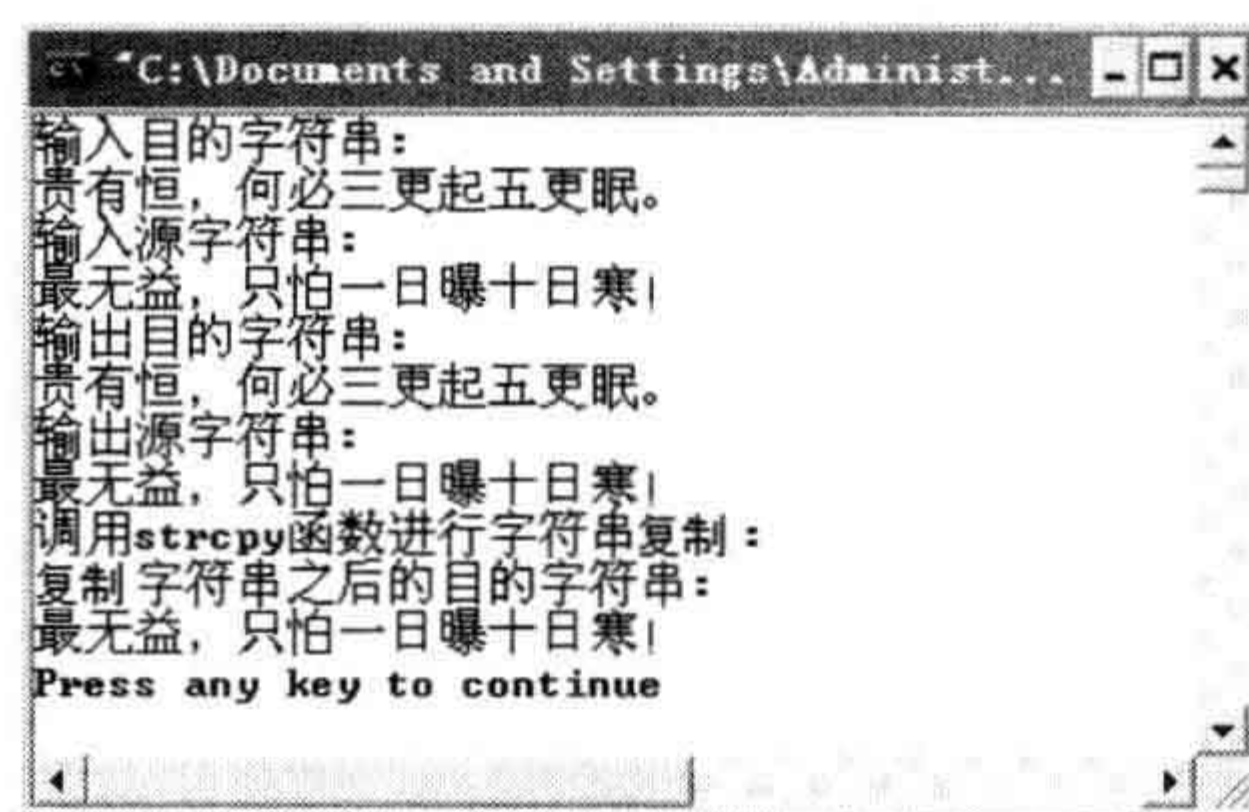


图 9.1 字符串复制

专家点评

字符串复制有以下几点注意事项:

- (1) 不能用赋值语句将一个字符串常量或字符数组直接赋给一个字符数组。
- (2) 目的字符数组应有足够的长度,否则不能全部装入所复制的字符串。也就是说,目的字符数组的长度一定不能比源字符数组的长度短。
- (3) “目的字符数组”必须写成数组名形式,而“源字符数组”可以是字符数组名,也可以是一个字符串常量,这时相当于把一个字符串赋予一个字符数组。

问题 144 如何进行字符数组的连接?

问题阐述

有两个字符数组 a 和 b, a 的值是“mingri”, b 的值是“book”,现在要把 b 连接在 a 的后边,使 a 变成“mingri book”,应该怎么做?

专家解答

字符串连接就是将一个字符串连接到另一个字符串的末尾,使其组合成一个新的字符





串，在字符串处理函数中，`strcat` 函数就具有字符串连接的功能。其语法格式如下：

`strcat`(目的字符数组名, 源字符数组名)

`strcat`()函数的功能是把源字符数组中的字符串连接到目的字符数组中字符串的后面，并删去目的字符数组中原有的串结束标志 ‘\0’。因此，目的字符数组必须拥有足够大的长度，以免发生因不能装下连接源字符数组后的新字符串，而产生错误。

使用 `strcat`()函数必须包含 `string.h` 头文件。

下面通过实例来介绍一下 `strcat`()函数的使用。

例如，在 `main()`函数体中定义两个字符数组，分别用来存储源字符串和目的字符数组，然后获取用户为两个字符数组赋值的字符串，并分别输出两个字符数组，调用 `strcat`()函数将源字符数组中的字符串连接到目的字符数组中字符串的后面，最后输出目的字符数组。具体代码如下。

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str1[100],str2[100];
    printf("输入目的字符串: \n");
    gets(str1);                                /*输入目的字符*/
    printf("输入源字符串: \n");
    gets(str2);                                /*输入源字符串*/
    printf("输出目的字符串: \n");
    puts(str1);                                /*输出目的字符*/
    printf("输出源字符串: \n");
    puts(str2);                                /*输出源字符串*/
    strcat(str1,str2);                          /*调用 strcat()函数进行字符串连接*/
    printf("调用 strcat 函数进行字符串连接: \n");
    printf("字符串连接之后的目的字符串: \n");
    puts(str1);                                /*输出连接后的目的字符串*/
    return 0;                                  /*程序结束*/
}
```

运行程序，字符串连接结果如图 9.2 所示。

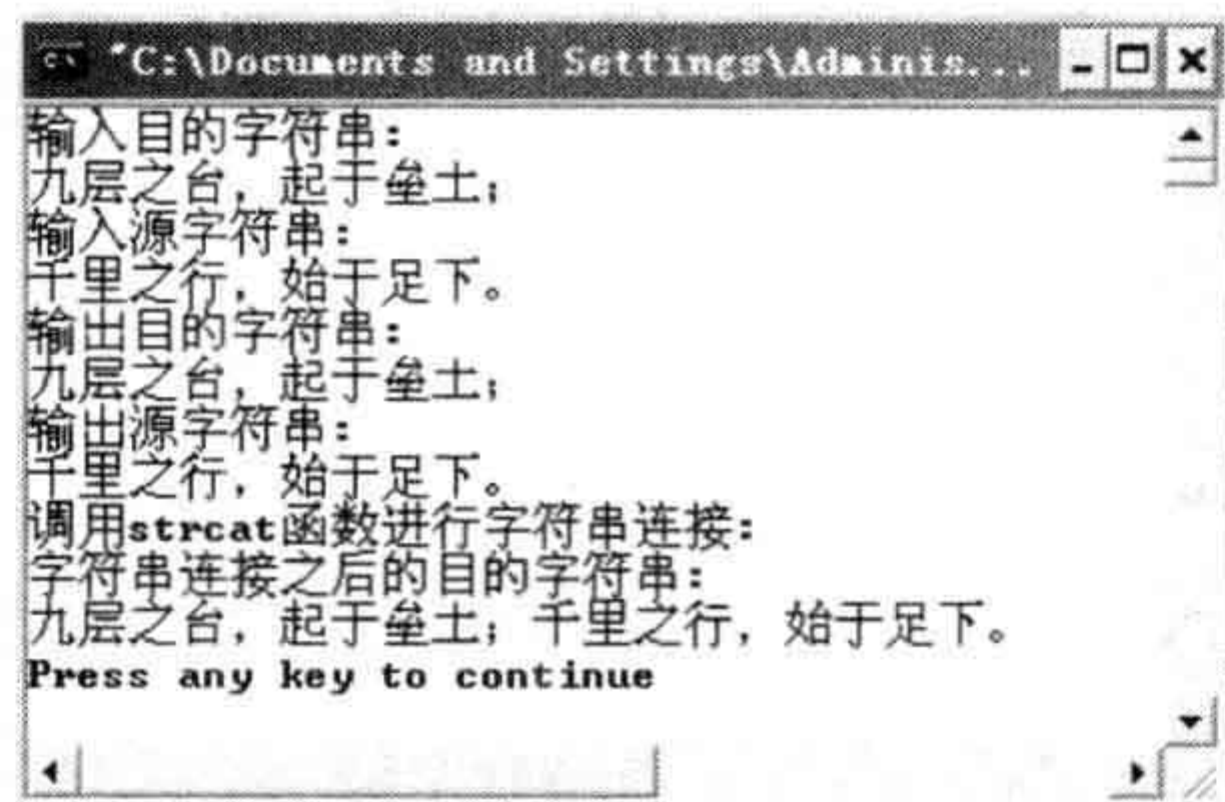


图 9.2 字符串连接



专家点评

字符串连接函数能把第二个字符串连接在第一个字符串之后,如果希望把一个字符串数组(二维字符数组)合成一个长字符串,可以用下面程序。

```
char strmerge[800]={};  
char strdiv[10][80]={"I","am","a","student"};  
for(i=0;i<4;i++)  
    strcat(strmerge,strsub[i]);
```



Note

问题 145 如何进行字符串的比较?

问题阐述

字符串的比较,就是看两个字符串哪个大,哪个小。比较的原则是对两个字符串进行逐个字符的比较。直到有不相等的字符为止。例如:

```
"abcd"  
"abxa"
```

这两个字符串哪个大呢?两个字符串前两个字符相同,第三个字符按 ASCII 比较 c 小于 x,因此,整个字符串“abcd”小于“abxa”。

以上是字符串比较的原则,那么 C 语言中要想做这样的比较,应该怎样去操作呢?

专家解答

字符串比较使用 strcmp()函数。其语法格式如下。

```
strcmp(字符数组名 1, 字符数组名 2)
```

strcmp()函数的功能:字符串比较就是将一个字符串与另一个字符串从首字母开始,按照 ASCII 码的顺序进行逐个比较,直到两个字符串的 ASCII 顺序不相等或者遇到结束标志“\0”时结束比较,并由函数返回值返回比较结果。

返回值如下:

- ☑ 字符串 1=字符串 2, 返回值为 0。
- ☑ 字符串 1>字符串 2, 返回值为正数。
- ☑ 字符串 1<字符串 2, 返回值为负数。

在 strcmp()函数字符串比较中,字符串 1 和字符串 2 不但可以是字符数组,还可以是字符串常量。

其实,strcmp()的结果就是当两个字符串进行比较的时候,若出现不同的字符,则以第一个不同的字符的比较结果作为整个比较的结果。

使用 strcmp()函数必须包含 string.h 头文件。

下面通过实例来介绍一下 strcmp()函数的使用。



例如，在 main() 函数体中定义 4 个字符数组，分别用来存储用户名、密码和用户输入的用户名及密码字符串，然后分别调用 strcmp() 函数比较用户输入的用户名和密码是否正确。具体代码如下。



Note

```
#include<stdio.h>
#include<string.h>
int main()
{
    char yhm[20]= {"mrkj"};           /*设置用户名字符串*/
    char mima[20] = {"111"};          /*设置密码字符串*/
    char yhm1[20],mima1[20];
    int i=0;
    while(i < 3)
    {
        printf("输入用户名字符串: \n");
        gets(yhm1);                   /*输入用户名字符串*/
        printf("输入密码字符串: \n");
        gets(mima1);                  /*输入密码字符串*/
        if(strcmp(yhm,yhm1))          /*如果用户名字符串不相等*/
        {
            printf("用户名字符串输入错误! \n"); /*提示用户名字符串输入错误*/
        }
        else                           /*用户名字符串相等*/
        {
            if(strcmp(mima,mima1))      /*如果密码字符串不相等*/
            {
                printf("密码字符串输入错误! \n"); /*提示密码字符串输入错误*/
            }
            else                         /*用户名和密码字符串都正确*/
            {
                printf("欢迎使用! \n");        /*输出欢迎字符串*/
                break;
            }
        }
        i++;
    }
    if(i == 3)
    {
        printf("输入字符串错误 3 次! \n"); /*输入字符串错误 3 次*/
    }
    return 0;                          /*程序结束*/
}
```

运行程序，字符串比较结果如图 9.3 所示。

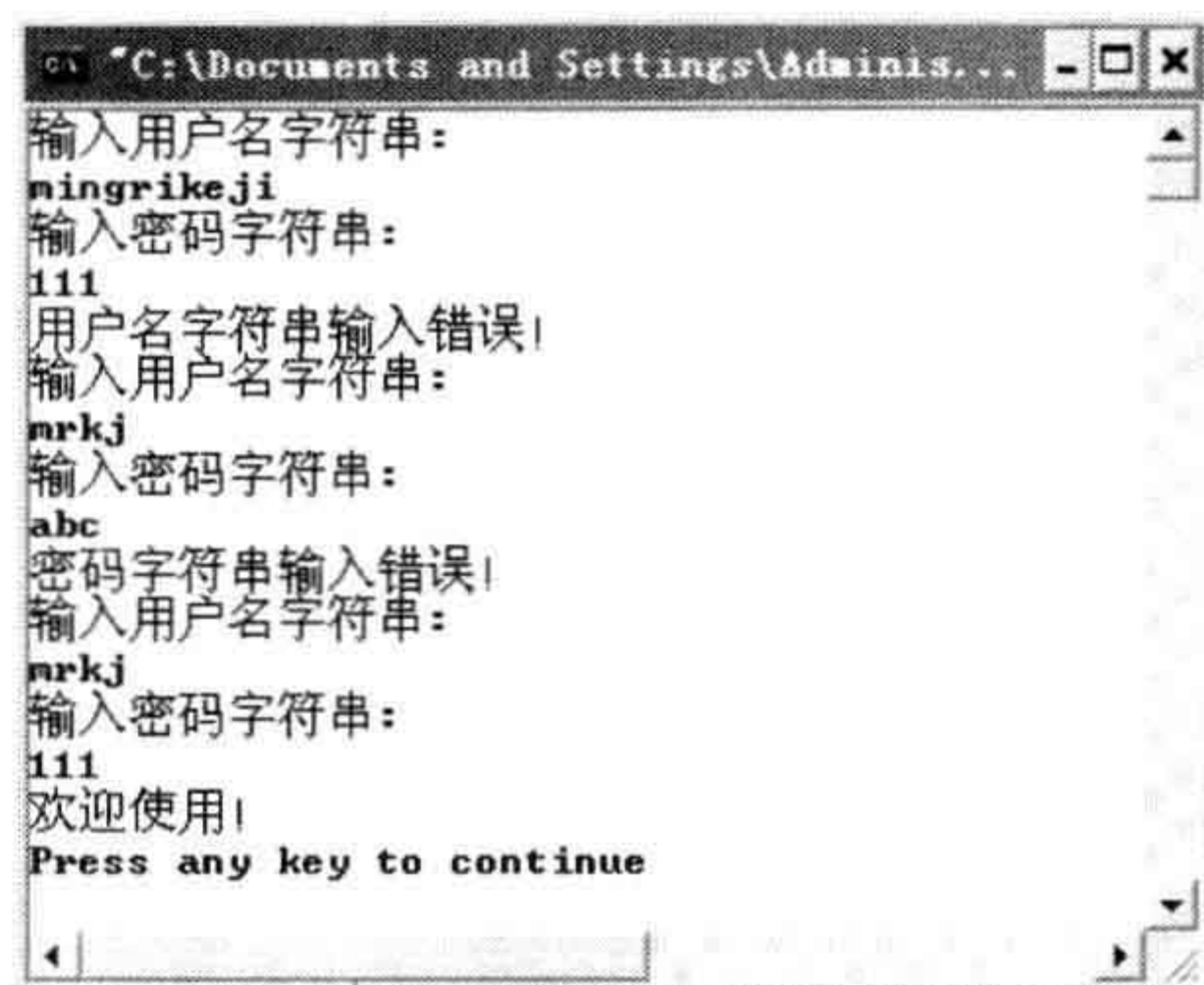


图 9.3 字符串比较

专家点评

字符串的比较不能用关系运算符>、<、==，只能用 strcmp()来实现。

问题 146 如何测定字符串的长度？

问题阐述

字符串的长度指字符串中有多少个字符，不指字符串占用内存的大小。例如：

```
char a[80]="abcd";
```

字符串占用内存空间是 80 个字节，而字符串长度是 4。

那么，怎样在程序中求出字符串的长度呢？

专家解答

通过循环来判断字符串结束标志'\0'，虽然也能获得字符串的长度，但是实现起来相对烦琐。这时，可以使用 strlen()函数来计算字符串的长度。strlen()函数的语法格式如下。

```
strlen(字符数组名)
```

strlen()函数的功能是计算字符串的实际长度（不含字符串结束标志'\0'），函数返回值为字符串的实际长度。

下面通过实例来介绍一下 strlen()函数的使用，代码如下。

```
#include<stdio.h>
#include<string.h>
int main()
{
    char c1[50],c2[50];
    int num;
    printf("输入一个字符串: \n");
    scanf("%s", &c1);
```

/*获取输入的字符串*/



Note

```

num = strlen(c1);                                /*计算字符串长度*/
printf("字符串的长度为: %d\n",num);              /*输出字符串长度*/
printf("再输入一个字符串: \n");
scanf("%s", &c2);                                /*获取输入的字符串*/
num = strlen(c2);                                /*计算字符串长度*/
printf("字符串的长度为: %d\n",num);              /*输出字符串长度*/
strcat(c1,c2);                                   /*连接字符串*/
printf("将两个字符串进行连接: \n%s\n",c1);       /*输出连接后的字符串*/
num = strlen(c1);                                /*计算连接后的字符串长度*/
printf("连接后的字符串长度为: %d\n",num);        /*输出连接后的字符串*/
return 0;                                        /*程序结束*/
}

```

(1) 在 main() 函数体中定义两个字符数组, 用来存储用户输入的字符串。

(2) 显示提示信息, 要求用户输入一个字符串, c1 获取用户输入的第一个字符串, 并使用 strlen() 函数将字符串的长度计算出来, 并使用 printf() 函数输出结果。

(3) 再次显示要求用户输入一个字符串的信息, c2 获取用户输入的第二个字符串, 并使用 strlen() 函数将字符串的长度计算出来, 并使用 printf() 函数输出结果。

(4) 使用 strcat() 函数, 将两个字符串 c1、c2 连接起来, 使用 strlen() 函数将连接后的字符串的长度计算出来, 并使用 printf() 函数输出结果。

运行程序, 获取字符串长度结果如图 9.4 所示。

程序是使用 VC++6.0 编译器进行编写的, 支持汉语输入。在 VC++6.0 编译器中, 每个汉字和汉语符号的字符串长度都为 2, 所以程序中输入第一个字符串的长度为 4, 字母和英文下的标点长度是 1, 第二个输入的字符串长度为 7。用中文输入法输入的标点符号字符串长度为 2, 用英文输入法输入的标点符号字符串长度为 1。

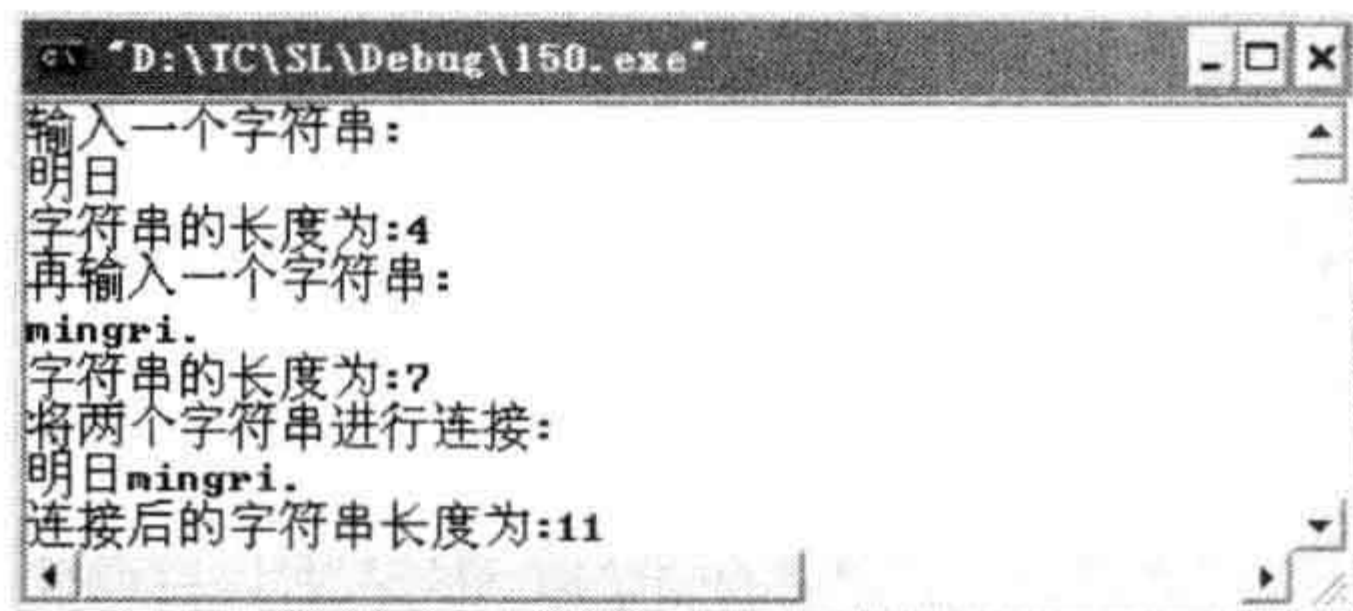


图 9.4 获取字符串长度

专家点评

字符串长度指字符个数, 不是在内存中占几个字节。

问题 147 如何进行字符串大小写的相互转换?

问题阐述

一个字符数组的值是 “MingRi!”, 现在要把所有字母变成大写或变成小写, 应怎样操作?

专家解答

字符串的大小写转换需要使用strupr() 函数和 strlwr() 函数。strupr() 函数的语法格式如下。



strupr (字符串)

strupr()函数的功能是将字符串中的小写字母变成大写字母,其他字母不变。

strlwr()函数的语法格式如下。

strlwr (字符串)

strlwr()函数的功能是将字符串中的大写字母变成小写字母,其他字母不变。

下面通过实例来介绍一下 strupr()函数和 strlwr()函数的使用方法。

例如,在 main()函数体中定义两个字符数组,分别用来存储要转换的字符串和转换后的字符串,然后根据用户输入的操作指令判断调用 strupr()函数或者 strlwr()函数进行大小写转换。具体代码如下。

```
#include<stdio.h>
#include<string.h>
int main()
{
    char text[20],change[20];
    int num;
    int i=0;
    while(1)
    {
        printf("输入转换大小写方式 (1 表示大写, 2 表示小写, 0 表示退出): \n");
        scanf("%d", &num);
        if(num == 1)                                /*如果是转换为大写*/
        {
            printf("输入一个字符串: \n");
            gets(text);                             /*输入要转换的字符串*/
            strcpy(change,text);                    /*复制要转换的字符串*/
            strupr(change);                          /*字符串转换大写*/
            printf("转换成大写字母的字符串为: \n%s\n",change);/*输出转换后的字符串*/
        }
        else if(num == 2)                          /*如果是转换为小写*/
        {
            printf("输入一个字符串: \n");
            gets(text);                             /*输入要转换的字符串*/
            strcpy(change,text);                    /*拷贝要转换的字符串*/
            strlwr(change);                         /*字符串转换小写*/
            printf("转换成小写字母的字符串为: \n%s\n",change);/*输出转换后的字符串*/
        }
        else if(num == 0)                          /*如果命令字符为 0*/
        {
            break;                                  /*跳出当前循环*/
        }
    }
    return 0;                                       /*程序结束*/
}
```



Note



运行程序，字符串大小写转换结果如图 9.5 所示。

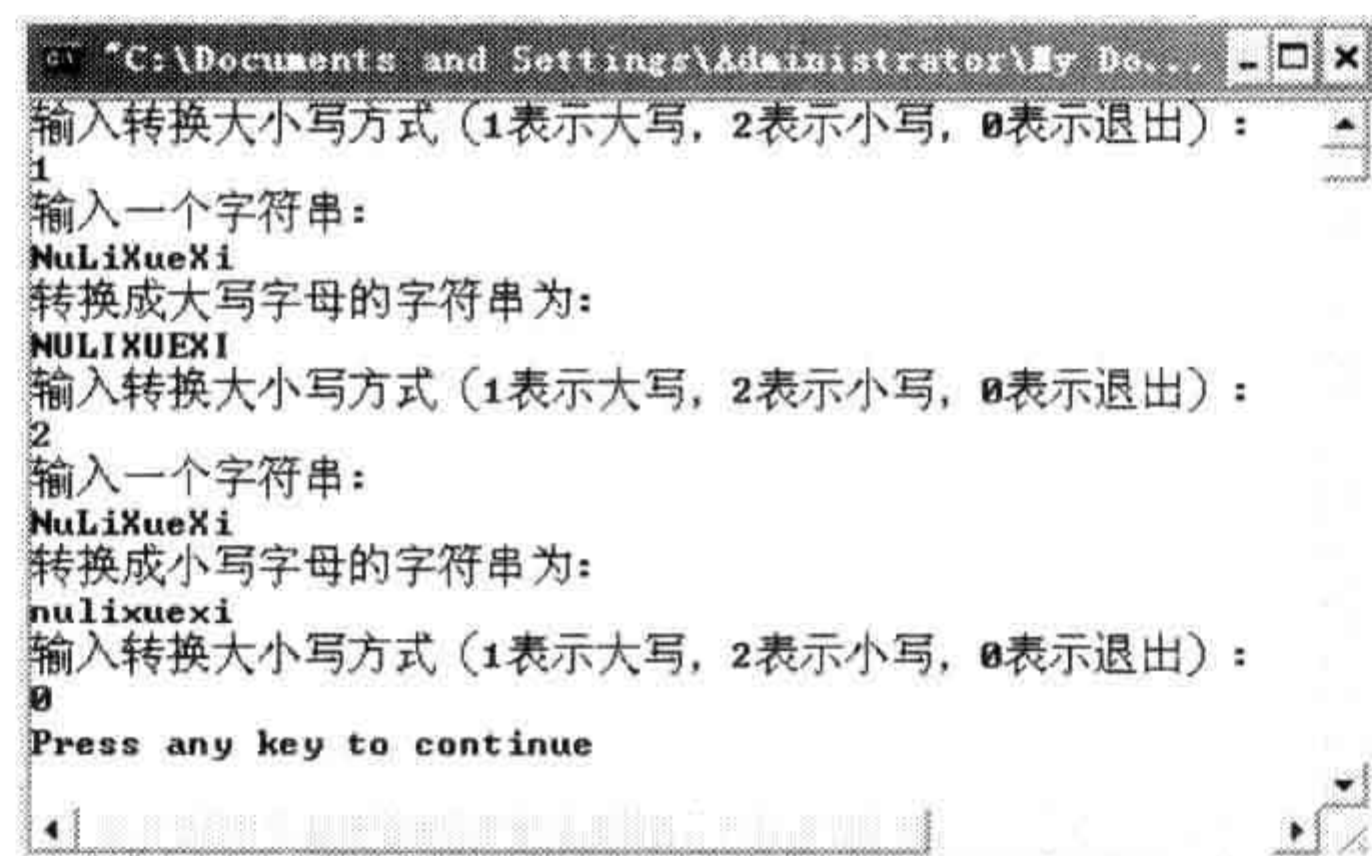


图 9.5 字符串大小写转换

专家点评

strupr()函数和 strlwr()函数可以对字符串中所有字母进行转换，如果要把每个单词首字母变成大写，只能自己编写程序，逐个字符去处理。

问题 148 如何计算字符串中有多少个单词？

问题阐述

有一个字符串 “You may not do any of the following.” 统计单词的个数。我们规定，连续字母或数字是一个单词，如 “b2b” 是一个单词，空格或其他任何字符分开的都不是一个单词。

专家解答

以下函数功能是计算一个字符串中单词的个数。程序中用 flag 记录上一位置是否是单词，如果上一位置不是字母而当前位置是字母，那就是一个新单词的出现，将变量 word 加 1。

```
#include<string.h>
#include<stdio.h>
#include<ctype.h>
int wordCount(char *str)
{
    int flag=0;                /*上一位置是否在单词中*/
    int word=0;                /*单词个数*/
    int i,n;
    n=strlen(str);             /*字符串长度*/
    for(i=0;i<n;i++)
    {
        if(flag==0 && isalnum(str[i])) /*前一位置不是字母，当前位置是字母*/
            word++;
    }
}
```



Note



```

        if(isalnum(str[i]))           /*当前位置是字母 flag=1,否则 flag=0, 用于下次的判定*/
            flag=1;
        else
            flag=0;
    }
    return word;
}
main()
{
    char str[] = "You may not do any of the following.";
    clrscr();
    printf("string \"%s\" word count is %d",str,wordCount(str));
}

```



Note

专家点评

计算字符串中单词个数有多种算法,请读者发挥自己的聪明才智,编写其他算法。

问题 149 gets()函数和 scanf()函数在输入字符串时有何区别?

问题阐述

gets()函数是专门用于字符串输入的函数,scanf()函数的%s 格式也可以输入字符串,二者是否完全相同?怎样区分使用这两个函数?

专家解答

scanf()函数用不同格式说明符号%d、%f、%s 等可以输入各种类型数据。输入数据时,空格、回车或 Tab 是各输入项的分隔符。如:

```

int a,b;
scanf("%d%d",&a,&b);

```

执行时输入 3□5, □表示空格,变量 a 得 3,变量 b 得 5。
同样,如下字符数组:

```

char a[80],b[80];
scanf("%s%s",a,b);

```

执行时输入 mingri□software, □表示空格,变量 a 得 mingri,变量 b 得 software。
注意,□本身也是一个字符。如果希望 a 的值是 mingri□software,该怎样输入呢?
scanf()函数实现不了这样的功能。此时使用 gets()函数:

```

gets(a);

```




当输入 mingri□software 时, a 的值即为 mingri□software。

gets()函数将一行输入中的所有字符都给一个变量,它也不允许一次为两个变量输入数据。

scanf()函数以空格、Enter 键或 Tab 键为分隔符,也就是输入字符中不允许包含以上三个字符。

gets()只以回车为输入结束标志。



Note

专家点评

这两个函数的区别在于 gets()可以输入含空格、tab 的字符串,scanf()不能。

二者选用的原则:当输入数据字符串字符个数少,不会出现空格、tab 时,用 scanf 较好,反之 gets()较好。

问题 150 puts()函数和 printf()函数在输出字符串时有何区别?

问题阐述

puts()函数是专门用于字符串输出的函数,printf()函数的%s 格式也可以输出字符串,二者是否完全相同?怎样区分使用这两个函数?

专家解答

(1) printf()函数可以输出各种类型,并且一次可以输出多项。puts()函数只能输出字符串,一次只能输出一个字符串。

(2) 在它们都只输出一个字符串的区别可以由下面的例子看出。

```
char a[80]="mingri ";
char b[80]="book";
printf("%s",a);
printf("%s",b);
```

输出结果为:

```
mingri book
```

如果把两个 printf 改为 puts, 即:

```
puts(a)
puts(b);
```

结果为:

```
mingri
book
```




printf()与 puts()两个函数的区别在于, puts()输出后具有自动换行的功能, 而 printf()函数只有输入“\n”才能换行, 即:

```
puts(a)相当于 printf("%s\n",a);
```

专家点评

puts()函数在输出字符串时,不具有 printf()函数无法完成的功能。这点与 gets()和 scanf()的区别不同, 所以只要会用 printf()函数, 就可以解决所有字符串输出问题。



Note

问题 151 数组与指针的区别是什么?

问题阐述

常听说数组实质就是指针, 是这样的吗?

专家解答

数组是一组数, 这组数用下标相区分, 例如:

```
int a[5]={1, 3, 5, 4, 8};
```

产生 5 个数, 这 5 个数是 a[0]、a[1]、...、a[4]。

指针是一个变量的地址。例如:

```
int *p;
int a=5;
p=&a;
```

p 是另一指针变量, 用于保存另一变量的地址。

这样说来, 指针与数组间没有任何联系。

按上面的定义 p 和 a, 如果再有如下语句:

```
p=&a[0];
```

这样, 由于数组在内存中连续分配内存, 如图 9.6 所示 (假设 a[0]地址为 2000, 整型占两个字节), p 的值是 2000, *p 为 1, 即 a[0]。

2000	1	a[0]
2002	3	a[1]
2004	5	a[2]
2006	4	a[3]
2008	8	a[4]

图 9.6 数组分配内存的方式



注意, $p+1$ 的值不是 2001。C 语言中 $p+1$ 的值是由 p 的定义来决定的, 定义时指向几个字节的变量, 指针加 1 就加几个字节。由于 p 的定义是 int^*p ; int 占两个字节, 因此 $p+1$ 就是两个字节, 是 2002, 正好就是 $a[1]$ 的地址。

因此:

$p+1$ 值为 2002, $*(p+1)$ 为 5 即 $a[1]$

$p+i$ 即是 $\&a[i]$, 而 $*(p+i)$ 即为 $a[i]$



Note

脚下留神

以上 int 占两个字节只是一种假设, 因为 int 所占字节数实际在不同系统中是不一样的, 但无论是两个字节还是四个字节, 得到的结论是一样的。

C 语言中规定, 数组名表示数组首地址, 即:

a 即为 $\&a[0]$

因此:

$a+1$ 即 $*a[1]$;

$*(a+1)$ 即 $a[1]$

$*(a+i)$ 即 $a[i]$

C 语言规定指针变量也可以表示成下标写法, 即:

$*(p+1)$ 可以写成 $p[1]$

到此为止, 数组的所有表示法 $a[i]$, $a+i$, $*(a+i)$ 都可以有对应的指针表示法 $p[i]$, $p+i$, $*(p+i)$, 于是结论“数组即指针, 指针即数组”好像就成立了。

其实这两种说法都是错的, 只能说明在表示法上, 二者可以有相同的方法。下面再对二者做一区分。

数组名表示数组首地址, 可以说数组名是数组的指针, 但它不是变量, 只是一个常量, 即不能对数组名 a 进行重新赋值。而指针变量是变量, 可以重新赋值。

程序中可以这样写:

```
p=&a[1];
p=p+1;
p++;
```

以上写法都是正确的。

```
a=p;
a=a+1;
a++;
```

以上三种写法试图对 a 的重新赋值都是错误的。

专家点评

指针与数组是两个完全不同的概念, 当用指针表示数组时, 二者都有下标表示法和*



号表示法，数组名是一数组的首地址，是常量，指针变量可以指向数组，是变量。



Note

问题 152 为什么作为函数形参的数组和指针可以互换？

问题阐述

以下两种定义形参的方法为什么是相同的？

```
void fun(int a[])  
{  
  
}
```

```
void fun(int *a)  
{  
  
}
```

专家解答

先给出一个主函数，用于调用上面的子函数：

```
main()  
{  
    int a[10];  
    fun(a);  
    ...  
}
```

调用语句中的 `fun(a)`，由于 `a` 是数组名，所以形参要定义成第一种形式 `int a[]`，但也可以认为调用语句中 `fun(a)`，`a` 是数组首地址（参见问题 151 数组与指针的区别），即 `a` 是 `a[0]` 的地址，即 `&a[0]`，这样形参就应该是第二种形式指针 `int *a`，不论主调函数的实参如何理解，它的形式是一样的，因此，两种形参的意义是一样的。

专家点评

其实，C 语言形参中的数组名实质上就是指针变量。不论其定义形式如何，它与函数内定义的数组名是不同的。

即：

```
fun(int a[10])  
{int b[10];  
    ...  
}
```

其中的 `int a[10]` 等价于 `int *a`；可以进行指针的各种运算，包括赋值运算。



但 `int b[10]` 就是数组名。



Note

问题 153 为什么数组名作参数传递给子函数时，子函数可以改变主函数中数组的值？

问题阐述

C 语言中，参数的传递方式是按值传递，即不论形参怎样改变，实参不被改变，这是在简单变量作参数时。为什么在数组名做参数时，子函数中数组元素改变后，主函数中的实参的数组元素也改变了呢？即：

```
fun(b[],int n)
{
    b[1]=10;
    n=1;
}
main()
{int a[10]={1,3,5,4,8},n=10;
  fun(a,n);
  printf("%d,%d",a[1],n);
}
```

上面程序结果是 10,10 怎么不是 3,10 呢？

专家解答

C 语言形参中的数组名实质上是指针变量（参见问题 152 为什么作为函数形参的数组和指针可以互换），也就是不论形参定义成 `int a[]`，还是 `int *a`，其实都是 `int *a`。

那么主函数传递给子函数的是什么呢？是不是把数组中的 10 个数告诉了子函数呢？其实一个数都没有传递给子函数，它只传递了 `a[0]` 的地址，那子函数是怎样知道每个数的呢？

如果主函数中 `a[0]` 的地址是 2000，

2000	1	a[0]
2002	3	a[1]
2004	5	a[2]
2006	4	a[3]
2008	8	a[4]

图 9.7 数组的地址

那么子函数中指针变量 `b` 的值是 2000，注意主函数中的 `a` 是数组名，子函数中的 `b` 是指针变量，不论形参是 `int *b`，还是 `int b[]`。



b+1 是 2002 (为什么不是 2001? 参见问题 151 数组与指针的区别是什么?)。

*(b+1)即为 3, 也就是主程序中的 a[1]。

因此, 当子程序中有 x=*(b+1)语句时, x 值为 3;

而*(b+1)在写法上又可以表示为 b[1] (参见问题 151 数组与指针的区别是什么?), 所以子程序中写:

x=b[1], 即使用了主程序中的 a[1]。

扩展开来即 b[i]即 a[i]。

上面的赋值语句反过来写就是:

```
*(b+1)=10;
```

b+1 是主程序中 a[1]的地址, *(b+1)=10, 即将 10 存入主程序中 a[1]处。而*(b+1)又可以表示为 b[1], 所以子程序中 b[1]=10 可以改变主程序中的 a[1]。

问题阐述中的参数 n 为什么不变呢, 这就是按值传递的含义。主程序中的变量 n 的值是 10, 10 传递给了子程序, 子程序中用另一个变量 n 来保存, 这两个变量占用不同内存空间, 子程序中的 n 在子程序执行后内存被释放。因此, 不论子程序中的怎样改变, 主程序中的 n 不会改变。

专家点评

数组名作参数, 提供了一个让子函数返回一组改变了的值得方法。

问题 154 C 语言中有动态数组吗?

问题阐述

数组定义时, 必须先说明其所占存储空间的大小, 这样对空间大小不易确定的问题就很难定义数组。如想用数组表示所有人的考试成绩, 人数只能在执行时才确定下来, 编程时无法确定人数, 虽然可以先估计一个很大的空间, 但这对使用空间差别很大的情况不太适用。怎样定义一个动态数组, 才能使它可以在执行时确定数组的大小呢?

专家解答

严格来讲, C 语言中没有动态数组, 但可以用动态内存分配函数来模拟动态数组的功能。如先输入人数, 再根据输入的人数动态分配一段正合适的空间。

calloc()函数可以完成此功能, 格式如下:

(void*)calloc(分配单元个数, 每个单元大小)

如果用一个整数表示一个人的考试成绩, 一共有 n 个人, 可以写成:

```
int *p;  
p=(int*)calloc(n,sizeof(int));
```



Note



使用 `calloc()` 函数后要用 `free()` 释放内存。

```
void free(void*)
```

使用这两个函数要包含头文件 `stdlib.h`。

完整示例如下：



Note

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int n,i;
    int *p;
    printf("输入人数: ");
    scanf("%d",&n);
    p=(int*)calloc(n,sizeof(int));          /*为 n 个人分配内存*/
    if(p==NULL)
    {
        printf("内存分配错误");
        exit(0);
    }
    for(i=0;i<n;i++)                        /*输入 n 个人成绩, 保存在以上动态分配的内存中*/
    {
        printf("输入第%d 个人成绩",i+1);
        scanf("%d",p+i);
    }
    printf("\n 输入的成绩为: \n");
    for(i=0;i<n;i++)                        /*显示动态内存中的数据*/
        printf("第%d 个人的成绩为%d\n",i+1,*(p+i));
    free(p);
}
```

程序运行结果如图 9.8 所示。

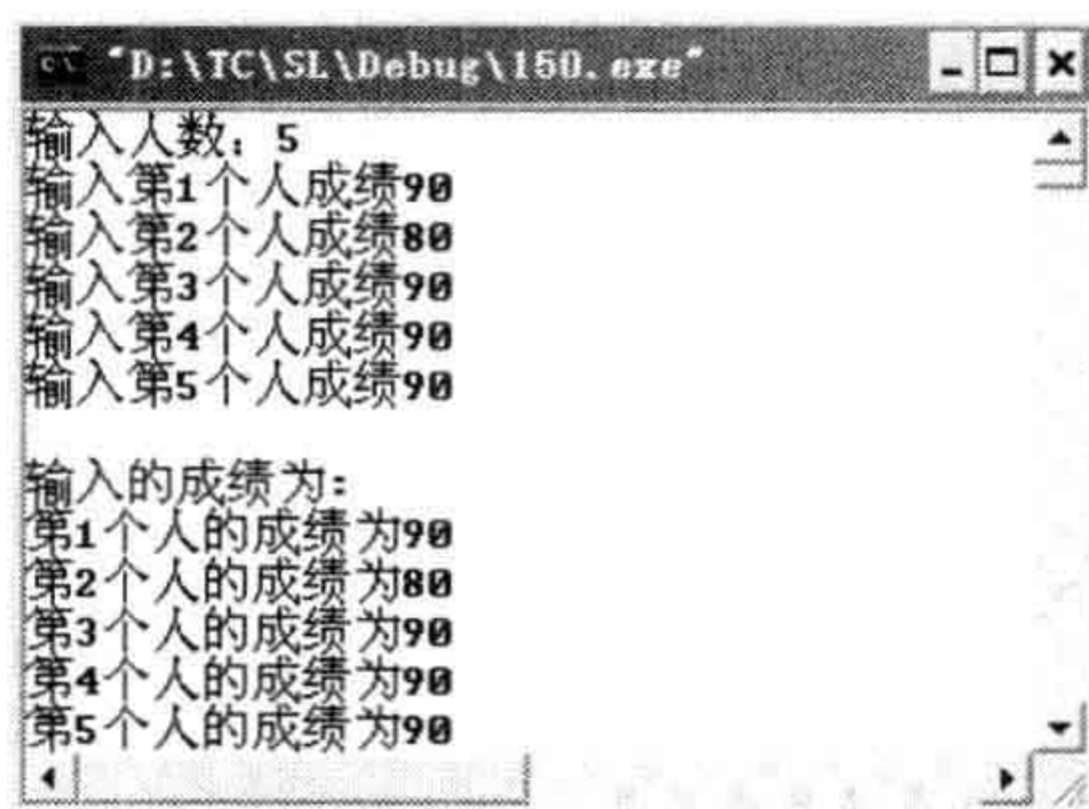


图 9.8 动态数组模拟

专家点评

C 语言中，可以用 `calloc()` 模拟实现动态数组的功能。



问题 155 如何实现动态二维数组?

问题阐述

有一个字符串,现在要以空格为分隔符,把它拆成多个子字符串,保存在一个类似二维数组中。由于无法预知可以分成多少个子字符串,因此想用动态二维数组,该怎么做呢?问题中说类似,因为每个字符串长度不一致,所以不是真正的二维数组。

专家解答

以上问题,如果希望每个子字符串占用空间不同,那就只能对每个子字符串都用动态分配的方式。用指针表示每个子字符串的地址,这样多个指针就得用指针数组,但指针数组定义时也要先定义数组大小。于是再用一次指针,该指针指向一组个数不一定的指针变量,即指针的指针。但指针的指针分配内存前那一组被指向的指针应该是大小固定的。

于是算法可以设计为先遍历整个字符串,确定能分成几个子字符串,再分配一组指针变量,用一个指针的指针指向其首地址,为这一组指针动态分配内存,保存第一个子字符串。

```
#include<stdio.h>
#include<stdlib.h>
char **splitStr(char*str,int *p)
{
    char **sub;
    int i,n=0,prei;
    for(i=0;str[i];i++)                /*计算有多少个逗号,即多少个子字符串*/
        if(str[i]==',')
            n++;
    if(str[i-1]!='')                    /*最后一个字符不是逗号,多加一个子字符串*/
        n++;
    sub=(char**)calloc(n,sizeof(char*)); /*sub 指向所有子字符串构成的列表,即指针数组的地址*/
    n=0;
    prei=0;                             /*子字符串开始位置*/
    for(i=0;str[i];i++)
        if(str[i]==',')                /*子字符串结束位置*/
        {
            sub[n]=(char*)calloc(i-prei+1,sizeof(char)); /*为每个子字符串分配内存*/
            memcpy(sub[n],str+prei,i-prei);               /*复制子字符串内容*/
            sub[n][i-prei]=0;                             /*为子字符串加上结束标志*/
            prei=i+1;                                     /*重新设置开始位置*/
            n++;                                           /*子字符串列表位置加 1*/
        }
    if(str[i-1]!='')                        /*最后一个字符不是逗号,剩下一个子字符串*/
```



Note



Note

```

    {
        sub[n]=(char*)calloc(i-prei+1,sizeof(char));
        memcpy(sub[n],str+prei,i-prei);
        sub[n][i-prei]=0;
        n++;
    }
    *p=n;
    return sub;
}
main()
{
    char str[100]="abc,de,fg,hi,jklm,,n";
    int n,i;
    char **p;
    p=splitStr(str,&n);
    for(i=0;i<n;i++)
        printf("%s\n",p[i]);
}

```

/*返回子字符串个数*/
/*返回子字符串列表首地址*/

第二种算法,针对本题的特点,不必为每个子字符串重新分配内存,就让那一组指针变量指向每个子字符串原来的位置。程序如下。

```

#include<stdio.h>
#include<stdlib.h>
char **splitStr(char*str,int *p)
{
    char **sub;
    int i,n=0;
    for(i=0;str[i];i++)
        if(str[i]==',')
            n++;
    if(str[i-1]!='.')
        n++;
    sub=(char**)calloc(n,sizeof(char*));
    n=0;
    sub[n++]=str;
    for(i=0;str[i];i++)
        if(str[i]==',')
        {
            str[i]='\0';
            sub[n]=str+i+1;
            n++;
        }
    *p=n;
    return sub;
}

```

/*计算有多少个逗号,即多少个子字符串*/
/*最后一个字符不是逗号,多加一个子字符串*/
/*sub 指向所有子字符串构成的列表,即指针数组的地址*/
/*第一个子字符串地址*/
/*给上一个子字符串加结束标志*/
/*把下一个子字符串地址保存起来*/
/*子字符串列表位置加 1*/
/*返回子字符串个数*/
/*返回子字符串列表首地址*/



```
}  
main()  
{  
    char str[100]="abc,de,fg,hi,jklm,,n";  
    int n,i;  
    char **p;  
    p=splitStr(str,&n);  
    for(i=0;i<n;i++)  
        printf("%s\n",p[i]);  
}
```

专家点评

动态二维数组要动态分配指针数组才能实现。

问题 156 strcpy()函数可以复制字符串的一部分吗?

问题阐述

有一个字符串“姓名：张三”，其中“姓名：”是固定格式，现在要取姓名的值放在另一变量中，可以用strcpy()函数吗？

专家解答

可以实现，这是一个对字符数组名即数组首地址的深入理解的问题。从最简单的开始，如下面的输出语句：

```
char a[80]="abcd";  
printf("%s",a);
```

输出结果是“abcd”。

a 是数组名，也是 a[0]的地址，上面的输出语句就是从&a[0]开始输出到结束标志\0 为止。那么：

```
printf("%s",a+1);
```

就是从&a[1]开始输出到结束标志\0 为止，结果为“bcd”。输出语句可以这样理解，其他语句也可以这样理解。

```
char a[80]="abcd";  
char b[80]="xyz";
```

strcpy(a,b);结果 a 变为“xyz”。

strcpy(a,b+1); 结果 a 变为“yz”。

strcpy(a+1,b+1); 结果 a 变为“ayz”。



strcpy(a+1,b); 结果 a 变为“axyz”。

在字符串处理函数中,所有用以数组名为参数的函数,如 strcpy()、strcat()、strcmp()、strlen()、printf()、scanf()、gets()、puts 都可以把数组名即首元素地址换成其他元素地址。

以上复制姓名的方法为:

```
char str[80]="姓名: 张三";
char name[20];
```

因为“姓名:”是固定格式,占 6 个字符(汉字和中文标点都是两个字符),所以 strcpy(name,str+6);就可以将姓名的值复制到 name 变量中。

再看一个单词逆置的例子。

有一个以空格为分隔符的字符串“this is a book”,以单词为单元将其逆置,变成“book a is this”,具体代码如下。

```
#include<stdio.h>
#include<string.h>
main()
{
    char a[80]="this is a book";
    char t[80]="";
    int n=strlen(a),i;
    for(i=n-1;i>0;i--)
        if(a[i]!=' ' && a[i-1]==' ')
        {
            strcat(t,a+i);
            a[i]='\0';
        }
    strcat(t,a+i);
    strcpy(a,t);
    printf("\n%s",a);
}
```

/*字符串长度*/
/*从后向前到第二个字符*/
/*到一个单词开始的位置,即一个字母的前面是空格*/
/*将该位置后的字符串连接到字符串 t*/
/*截后面的字符*/
/*将字符串开头剩下的一个单词连接到字符串 t 中*/
/*将结果从 t 复制回 a 中*/

专家点评

深入理解数组名即数组首地址,对于连续字符串的处理都可以使用字符串处理函数完成。

问题 157 字符串和字符数组有什么区别?

问题阐述

如题,字符串和字符数组有什么区别?



专家解答

在不严格的情况下，可以说字符数组就是字符串。

严格来讲，字符数组强调数组，是多个个体的集合；字符串是由多个个体构成的一个整体。

作为整体操作的字符串，能作为整体的基础是因为它的最后有一个结束标志‘\0’，因此，可以说有结束标志的一维字符数组就是字符串。

我们常说字符串处理函数，却从来不说字符数组处理函数，因为没有结束标志的字符数组是不能使用这组函数的。

如果人的姓名只由 26 个字母组成，现有一个字符数组有 26 个字母，它们是所有姓名字符的来源，我们可以说姓名中的每个字母来自一个字符数组，但不能说来自一个字符串，但姓名是一个字符串。这就是整体和个体集合的问题。姓名是一个整体，放在一起才有意义，而来源字符串的集合放在一起也没有独立意义。

字符串是一维的，字符数组可以是二维的、多维的。尽管可以有字符串数组，在字符串数组中，单个字符串仍然是一维的。

专家点评

C 语言中没有字符串这种数据类型，它是由一维字符数组加上结束标志构成的。

问题 158 ‘\0’和“\0”有什么区别？

问题阐述

一个是单引号的\0，一个是双引号的\0，二者有何区别？怎样应用？

专家解答

‘\0’是一个字符，数据类型是字符型。可以定义为：

```
char a='\0';
```

“\0”是一个字符串，数据类型是字符数组，“\0”可以与“ ”等价。可以定义为：

```
char a[2]="\0";
```

‘\0’是字符串的结束标志，字符串中在‘\0’后的所有字符认为是不存在的。

“\0”是空白字符串，该字符串中一个字符都没有。

在下面的字符串中：

```
char a[80]="abcd";
```

如果有赋值语句 `a[0]='\0'`；相当于字符串 `a` 是空白字符串“\0”或“ ”，因为第一个字符就是结束标志。



Note



专家点评

它们一个是单个字符，一个是字符串。



Note

问题 159 字符数组占用内存怎样算？

问题阐述

有三个字符数组：

```
char a[]="abcd";  
char b[]={ 'a', 'b', 'c', 'd' };  
char c[10]= "abcd";
```

它们各占用内存多少字节？

专家解答

a 占 5 个字节，因为除了 4 个字符，系统还会自动分配一个 ‘\0’ 。

b 占 4 个字节，如果字符数组以数组方式初始化，而不是以字符串方式初始化，那么最后没有那个 ‘\0’ 。

c 占 10 个字节。前 4 个字节是 “abcd”，第 5 个字节是 ‘\0’，后 5 个字节无定义。

专家点评

以上算的是占用内存的大小，如果计算字符串长度，以上三个字符串长度都是 4。计算占用内存空间用 sizeof(变量名)，计算字符串长度用 strlen(变量名)。

问题 160 用字符数组和指针两种方式定义的字符串有什么不同？

问题阐述

```
char a[]="abcd";  
char *b="abcd";  
char *p="xyz";
```

基于以上定义，b=p;没有错误，而 a=p 为什么就不正确？用字符数组和指针两种方式定义的字符串有什么不同？



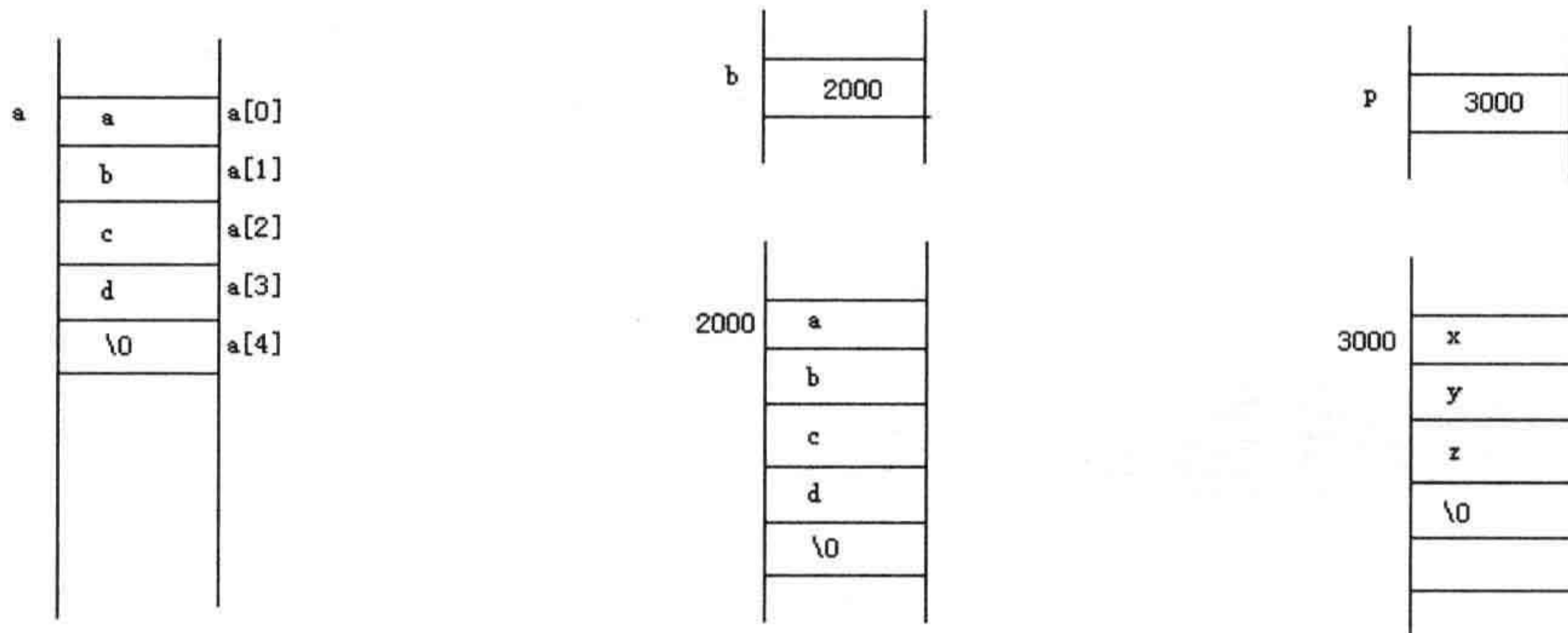
专家解答

a 定义的是一个数组，数组中有四个字符，这四个字符是 a[0], a[1], a[2], a[3], a 是数组名。

b 定义的是一个指针变量。变量的值是“abcd”这个字符串的首地址。可以通过图 9.9 说明两者的不同。



Note



(a) 数组 a 的定义

(b) 指针变量 b 的定义

(c) 指针变量 p 的定义

图 9.9

数组名是一个常量，不可被重新赋值，指针变量可以重新赋值。

当执行 `b=p` 时，b 的值变成“xyz”的地址 3000，而 `a=p` 是语法错误。

专家点评

注意，上述 `b=p` 与 `strcpy(b,p)` 虽然都能使 b 变成“xyz”，但内存中数据变化是不一样的。

`strcpy(b,p)` 函数把地址 3000 中的数据复制到地址 2000 中，而 `b=p` 的操作并没有改变地址 2000 中的数据，只是把指针变量 b 的值改成了 3000 即“xyz”的地址。

第10章

函数编程基础

- ▶▶ 什么是函数？如何分类？
- ▶▶ 如何定义无参函数？
- ▶▶ 如何定义有参函数？
- ▶▶ 什么是空函数？作用是什么？
- ▶▶ 什么是形参和实参？如何使用？
- ▶▶ 如何从函数返回？
- ▶▶ 函数返回值你了解多少？
- ▶▶ 如何进行函数的一般调用？
- ▶▶ 函数调用的基本方式有几种？各是什么？
- ▶▶ 函数调用应具备哪些条件？
- ▶▶ 如何进行函数的嵌套调用？
- ▶▶ 什么是递归调用？如何实现？
- ▶▶ 函数如何将数组元素作为实参？
- ▶▶ 如何将数组名作为函数参数？
- ▶▶ 如何将多维数组名作为函数参数？
- ▶▶ 什么是局部变量？
- ▶▶ 什么是全局变量？如何应用？
- ▶▶ 存储方式有几种？分别是什么？
- ▶▶ 如何使用 auto 关键字？
- ▶▶ 什么是静态变量？如何实现？
- ▶▶ 什么是寄存器变量？如何实现？
- ▶▶ 如何声明外部变量？
- ▶▶ 如何调用编译后的函数？
- ▶▶ 如何限定外部变量的使用范围？
- ▶▶ 如何使用函数调用实现对字符串的统计？
- ▶▶ main() 函数有什么作用？
- ▶▶ 什么是内部函数？
- ▶▶ 什么是外部函数？怎么用？
- ▶▶ static() 函数和普通函数有什么区别？
- ▶▶ 形参和实参有什么区别？



问题 161 什么是函数？如何分类？

问题阐述

一个 C 语言的源程序是由一个或者多个函数组成的。那到底什么是函数？怎么分类？

专家解答

函数是程序实现模块化编程的基本单元，一般是为了完成某一特定的功能，相当于其他语言中的子程序。

在其他编程语言中，一个较大程序的各项功能都是由其各个子程序来共同完成的。同样，C 程序的全部工作都是由各式各样的函数完成的。正因为此，C 语言也被称为函数式语言。由于采用了函数模块式的结构，C 语言易于实现结构化程序设计。使程序的层次结构清晰，便于程序的编写、阅读、调试。

1. 库函数与用户自定义函数

从函数定义的角度来看，在 C 语言中主要有两种函数，一种是库函数，另一种是用户自定义函数。

☑ 库函数：由 C 系统提供，用户无须定义。在调用库函数时也不必在程序中进行类型声明，只需在程序前包含该函数原型的头文件，即可在程序中直接调用。例如，在调用 `scanf()` 函数和 `printf()` 函数之前，应在程序开始部分包含 `stdio.h` 这个头文件。又如，调用字符串操作函数 `strlen()`、`strcmp()` 等时，也应在程序开始部分包含 `string.h`。

- I/O 函数：用于完成输入/输出功能。
- 数学函数：用于数学计算。
- 时间转换和操作函数：用于日期、时间转换操作。
- 字符屏幕和图形功能函数：用于字符屏幕管理和各种图形绘制功能。
- 字符串函数：用于字符串操作和处理。
- 目录路径函数：用于文件目录和路径操作。
- 动态地址函数：用于从自由内存区中分配所需地址空间。
- 接口函数：用于与操作系统最内层连接。
- 内存函数：用于内存管理、读取等操作。
- 过程控制函数：用于控制程序执行、终止等。
- 其他函数：这些函数不能简单地归属某一类，但都各具功能。

☑ 用户自定义函数：就是用户自己编写的用来实现特定功能的函数。

例如，计算任意两个整数的积，代码如下。

```
#include <stdio.h>
int mul(int x,int y)
```

```
/*自定义求积函数*/
```



Note



Note

```

{
    int z;
    z=x*y;
    return z;                                /*将所求的积返回*/
}
main()
{
    int a,b,c;
    printf("please input a and b:\n");
    scanf("%d,%d",&a,&b);
    c=mul(a,b);                               /*调用 mul()函数*/
    printf("the product is:%d\n",c);
}

```

程序运行结果如图 10.1 所示。

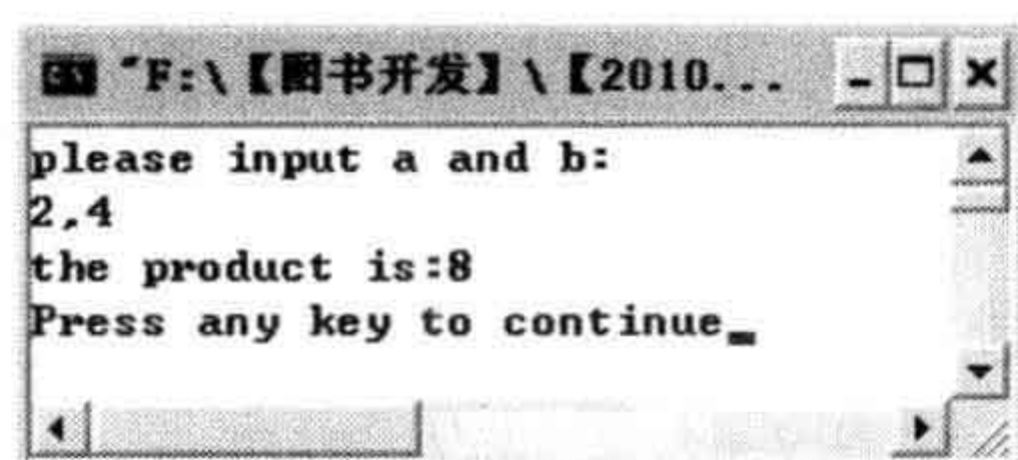


图 10.1 计算乘积

上述程序中的 `mul()` 函数就是用户自定义函数，它所要实现的功能就是计算出两数相乘的结果。

2. 有参函数和无参函数

从函数的形式上看，函数分为有参函数和无参函数两种。

- ☑ 有参函数：即在调用函数时，在主调函数和被调用函数之间有数据传递。如上述代码中的 `mul()` 函数就是有参函数，在主调函数 `main()` 和被调用函数 `mul()` 之间传递的数据就是 `a` 和 `b`。
- ☑ 无参函数：同有参函数相反，即调用无参函数时，主调函数并不将数据传送给被调用函数。

3. 有返回值函数和无返回值函数

C 语言的函数兼有其他语言中的函数和过程两种功能，从这个角度看，又可将函数分为有返回值函数和无返回值函数两种。

- ☑ 有返回值函数：被调用执行完后将向调用者返回一个执行结果，称为函数返回值。例如，数学函数就属于此类函数。由用户定义的这种要返回值的函数，必须在函数定义和函数声明中明确返回值的类型。
- ☑ 无返回值函数：用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。这类函数类似于其他语言中的过程。由于函数无须返回值，用户在定义此类函数时可指定其返回值为“空类型”，空类型的声明符为 `void`。

4. `main()` 函数

在上述代码中有一个 `main()` 函数，这个 `main()` 函数是由系统预定义的。C 程序的执行



从 `main()` 函数开始，在调用完其他函数后流程返回到 `main()` 函数，在 `main()` 函数中结束整个程序的运行。

C 程序中所有函数都是平行的，即在定义函数时是互相独立的。在一个函数中不能嵌套定义另一个函数，但是函数间可以相互调用（有一个例外，`main()` 函数是不能被调用的）。



Note

专家点评

函数的灵活应用对于程序项目的开发和维护都有着不可替代的功效，希望读者对此充分认识，谨慎以待。

问题 162 如何定义无参函数？

问题阐述

无参函数就是不需要传入参数，那么如何定义无参函数呢？

专家解答

无参函数的一般形式如下。

```
类型声明符 函数名()
{
    声明部分;
    语句;
}
```

类型说明符指明了本函数的类型，函数的类型实际上是函数返回值的类型。该类型声明符与前面介绍的各种声明符相同。在很多情况下，都不要求无参函数有返回值，此时函数类型声明符可以写为 `void`。函数名是由用户定义的标识符。函数名后有一个空括号，其中并无参数，但括号不可少。{} 中的内容称为函数体。

例如，下面定义了一个无参函数 `hello()`。

```
void hello()
{
    printf("hello mingri!");
}
```

专家点评

在一些项目中，功能菜单选项的输出等对无参函数的使用是比较频繁的，所以无参函数也是函数的重要组成部分。



问题 163 如何定义有参函数？

问题阐述

有参函数是函数的重点部分，那么如何定义有参函数呢？

专家解答

有参函数定义的一般形式如下。

```
类型声明符 函数名(形式参数列表)
{
    声明部分;
    语句;
}
```

在形参列表中给出的参数称为形式参数，它们可以是各种类型的变量，同时要对这些变量给予类型声明，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。

下面定义一个有参函数实现两数相加求和，并将求出的和作为返回值返回。具体代码如下。

```
int add(int x,int y)
{
    int sum;
    sum=x+y;
    return sum;
}
```

第一行说明 add() 函数是一个整型函数，其返回值是一个整数。形参为 x 和 y，这里也分别对 x 和 y 进行了类型声明，均为基本整型。x 和 y 的具体值是由主调函数在调用该函数时传送过来的。在 {} 中的函数体内，除形参外还定义了一个变量 sum，该变量仍为基本整型。函数体中的 return 语句是把 sum 的值作为函数的值返回给主调函数。有返回值的函数中至少应有一条 return 语句。

专家点评

如果在定义函数时不指定函数类型，那么系统会隐含指定函数类型为 int 类型。

问题 164 什么是空函数？作用是什么？

问题阐述

空函数在当前开发阶段也许没什么作用，但是在代码合并和日后的维护中却起着重要



的作用。那么什么是空函数？有何作用呢？

专家解答

空函数的一般形式如下。

```
类型声明符 函数名()  
{  
}
```



Note

空函数什么也不做，没有什么实际作用。既然没有什么实际功能，那为什么要存在呢？原因是空函数所处的位置是要放一个函数的，只是这个函数现在还未编好，用这个空函数先占一个位置，以后用一个编好的函数来取代它。

专家点评

在程序设计中往往根据需要确定很多模块，而这些模块就是由一些函数来实现。但是对于一些不确定的功能，可以使用空函数进行占位。

问题 165 什么是形参和实参？如何使用？

问题阐述

在调用函数的时候，调用函数和被调用函数之间基本都会发生数据传递关系，这就要用到有参函数，而不传递数据的就需要无参函数。这样必然涉及形参和实参的概念，那么究竟什么是形参？什么又是实参呢？如何应用？

专家解答

定义函数 `int mul(int x,int y)` 时，函数名后面括号中的变量名就是“形式参数”（简称形参）。例如，问题 161 代码中自定义的乘积函数 `mul()` 中的 `x` 和 `y` 就是形式参数。实际参数就是在主调函数中调用一个函数时，函数名后面括号中的参数。例如，问题 161 代码中的 `a` 和 `b` 就是实际参数。

形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是进行数据传送。发生函数调用时，主调函数把实参的值传送给被调函数的形参，从而实现主调函数向被调函数的数据传送。

函数的形参和实参具有以下特点：

（1）形参变量只有在被调用时才分配内存单元，在调用结束时将释放所分配的内存单元。因此，形参只在函数内部有效。函数调用结束返回主调函数后，就不能再使用该形参变量了。



Note

(2) 实参可以是常量、变量、表达式、函数等。无论实参是何种类型的量,在进行函数调用之前,每个实参都必须具有确定的值,以便把这些值传递给形参(如果形参是数组名,则传递的是数组首地址而不是数组的值,这点会在后面提到)。因此,应预先用赋值、输入等方法使实参获得确定值。

(3) 实参和形参的类型应相同或赋值兼容。例如下面的形式。

```
int mul(int x,int y)
{
    int z;
    z=x*y;
    return z;
}
main()
{
    float a,b,c;
    printf("please input a and b:\n");
    scanf("%f,%f",&a,&b);
    c=mul(a,b);
    printf("the product is:%f",c);
}
```

当通过键盘输入 2.5, 3.8 时, 程序运行结果如图 10.2 所示。

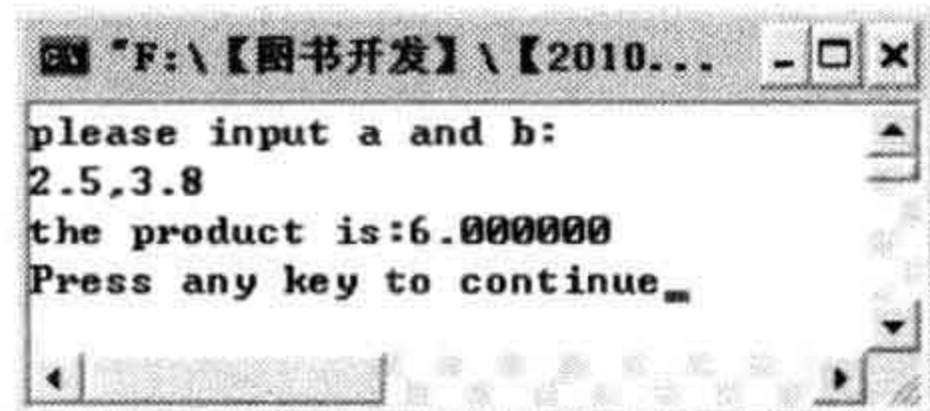


图 10.2 程序运行结果

通过图 10.2 会发现 2.5 与 3.8 的积是 6, 显然这个结果不正确。这是为什么呢? 因为形参的数据类型是基本整型, 而实参的数据类型是单精度型, 实参和形参的数据类型不同, 所以最终结果产生了误差。

(4) C 语言规定, 实参变量对形参变量的数值传递是单向传递, 即只能由实参传给形参, 而不能由形参再传给实参。下面的程序可以说明这个问题。

计算函数 $f(x) = \begin{cases} x+10 & x>0 \\ x+20 & x<0 \\ x & x=0 \end{cases}$ 的值, 代码如下。

```
main()
{
    int n;
    printf("input number\n");
    scanf("%d",&n);
    f(n);
}
```

/*调用 f()函数*/



Note

```

int f(int n)                                /*自定义 f()函数来求相应的函数值*/
{
    int i;
    if(n>0)                                /*如果 n 大于 0, 则 n+10*/
        n=n+10;
    else
        if(n<0)                            /*如果 n 小于 0, 则 n+20*/
            n=n+20;
        else                                /*如果 n 等于 0, 则 n 值为 100*/
            n=100;
    printf("n=%d\n",n);                    /*输出 n 的值*/
}

```

程序运行结果如图 10.3 所示。

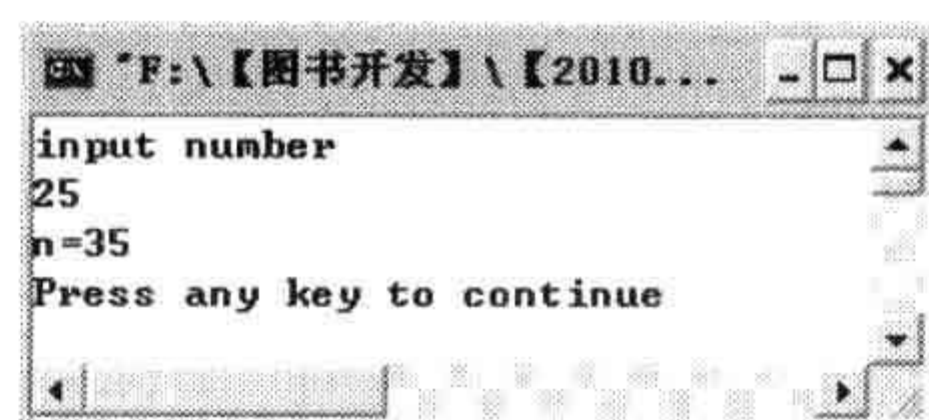


图 10.3 计算函数值

本程序中定义了一个函数 `f()`，该函数的功能是根据输入数的正负值不同与不同的数相加求和。在主函数中输入 `n` 值，并作为实参，在调用时传送给 `f()` 函数的形参 `n`。在函数 `f()` 中先用 `printf` 语句输出了一次 `n` 值，这个 `n` 值是形参最后取得的 `n` 值。在主函数中再用 `printf` 语句输出一次 `n` 值，这个 `n` 值是实参 `n` 的值。从运行情况看，输入的 `n` 值为 15，即实参 `n` 的值为 15。把此值传给函数 `f()` 时，形参 `n` 的初值也为 15。在执行函数过程中，形参 `n` 的值变为 25。返回主函数之后，输出实参 `n` 的值仍为 15。由此可见，实参的值不随形参的变化而变化。有一点要说明的是，这里的主函数和函数 `f()` 中用到的 `n` 应加以区别，这两个 `n` 不是同一个 `n`，它们各自作用的范围不同。

专家点评

对于到底是使用有参函数还是使用无参函数，需要根据函数所要实现的功能来决定。

问题 166 如何从函数返回？

问题阐述

函数被调用，但总是要终止执行，那么如何从函数返回呢？

专家解答

在编写程序的过程中，当要终止函数的执行，并返回到它的调用语句，许多时候会靠 `return` 语句来实现。使用 `return` 语句是为了返回一个值，或者是为了简化代码，通过设置



多个返回点来提高效率。程序代码如下。



Note

```
int ss(int i)
{
    int j;
    if (i <= 1)
        return 0;           /*如果 i 小于 1，则返回 0*/
    if (i == 2)
        return 1;           /*如果 i 等于 2，则返回 1*/
    for (j = 2; j < i; j++)
    {
        if (i % j == 0)
            return 0;        /*i 如果能被整除，则返回 0*/
        else if (i != j + 1)
            continue;
        else
            return 1;
    }
}
```

通过上面的代码可以发现一个函数中可能有多条返回语句。

专家点评

对于一些没有返回语句的函数，在执行完毕后，系统会默认地返回一个 0 值。具体内容将在问题 167 中详细介绍。

问题 167 函数返回值你了解多少？

问题阐述

在编写程序的过程中，当要终止函数的时候，会有一个返回值。那么函数返回值是什么样的呢？

专家解答

除了被定义为 void 类型的函数外，所有函数都返回一个值。这个值由 return 语句明确地给出。如果没有 return 语句，就返回 0。

在编写程序的过程中，通常会遇到 3 种类型的函数。

- ☑ 第一种函数只作单纯的计算，它们专门用于对指定的参数进行计算，并将结果返回。
- ☑ 第二种函数返回操作信息，并且返回一个表明操作是否成功的简单值。
- ☑ 第三种函数没有明确的返回值。



下面来看一个返回值的例子。求任意两个数的平均数，代码如下。

```
int ave(int a,int b)                                /*自定义求平均值函数*/
{
    int c;
    c=(a+b)/2;
    return c;                                       /*将所求平均值返回*/
}
main()
{
    int x,y,z;
    printf("please input x,y:\n");
    scanf("%d,%d",&x,&y);                          /*输入两个整数赋予 x 和 y*/
    z=ave(x,y);                                    /*调用 ave()函数*/
    printf("%d\n",z);
}
```



Note

程序运行结果如图 10.4 所示。

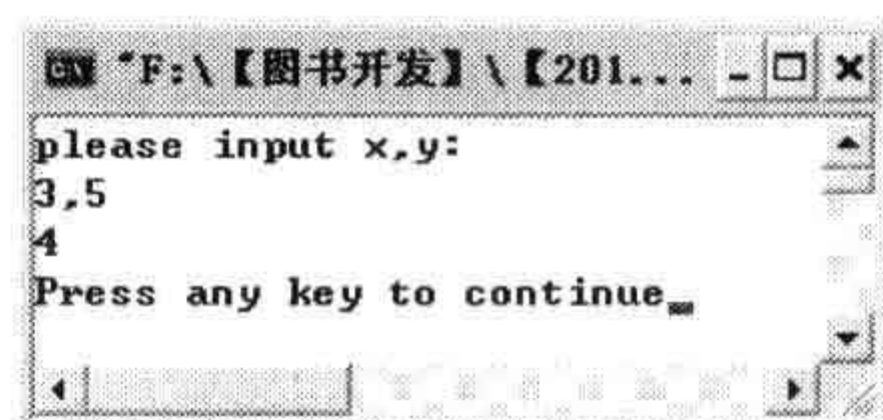


图 10.4 求平均数

专家点评

函数的返回值可以赋值，也可以不赋值。

问题 168 如何进行函数的一般调用？

问题阐述

函数调用的一般样式都是一样的，简称为函数的一般调用，那么函数的一般调用的形式是什么呢？

专家解答

在 C 语言中，函数调用的一般形式如下。

函数名(实际参数表)

对无参函数调用时，则无实际参数表。实际参数表中的参数可以是常数、变量或其他构造类型数据及表达式，各实参之间用逗号分隔。



专家点评

在此给出了基本的形式和使用方法，具体的使用规则要根据具体的编译环境来决定。关于通用规则，下面的问题中作了详细介绍。



Note

问题 169 函数调用的基本方式有几种？各是什么？

问题阐述

根据函数在程序中出现位置的不同，函数调用的方式也是不同的。那么其基本调用方式有几种？各是什么？

专家解答

函数调用的基本方式主要有以下 3 种。

(1) 函数语句。把函数调用作为一条语句。这样的函数调用方式，只要求函数完成一定的操作，不需要返回值。

(2) 函数表达式。函数调用出现在一个表达式中，这种调用称做函数表达式调用。此时调用的函数需要具有函数的返回值，这个函数的返回值需要参与表达式的计算。

例如：

```
x=min(a,b) + max(c,d);
```

(3) 函数参数。函数调用的另外一种方式是可以作为另一个函数的参数调用。

例如：

```
x = max(a,min(b,c));
```

函数 min() 是作为函数 max() 的一个参数调用的，此时要求函数 min() 必须具有返回值，且类型应与函数 max() 的参数类型一致。

专家点评

对于函数的调用方式，并不是说一定就是这 3 种中的一种，有时可能是两种或三种混合使用的。因此，读者要熟练掌握这 3 种方式，以便更灵活地应用。

问题 170 函数调用应具备哪些条件？

问题阐述

在主调函数中调用某函数之前应对该被调函数进行说明（声明），这与使用变量之前



要先进行变量声明是一样的。那么函数调用应具备哪些条件呢？

专家解答

在主调函数中对被调函数进行声明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值进行相应的处理。

其一般形式为：

类型声明符 被调函数名(类型 形参, 类型 形参...);

或者：

类型声明符 被调函数名(类型, 类型...);

括号内给出了形参的类型和形参名，或只给出形参类型。这便于编译系统进行检错，以防止可能出现的错误。

C 语言中又规定，在以下几种情况下可以省去主调函数中对被调函数的声明。

- ☑ 如果被调函数的返回值是整型或字符型，可以不对被调函数进行声明，而直接调用。这时系统将自动对被调函数返回值按整型处理。
- ☑ 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再作声明而直接调用。
- ☑ 如在所有函数定义之前，在函数外预先声明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数进行声明。例如：

```
char str(int a);  
float f(float b);  
main()  
{  
...  
}  
char str(int a)  
{  
...  
}  
float f(float b)  
{  
...  
}
```

其中第一、二行对 `str()` 函数和 `f()` 函数预先作了声明，因此在以后各函数中，无须对 `str()` 和 `f()` 函数再作声明就可直接调用。

专家点评

对库函数的调用不需要再作声明，但必须把该函数的头文件用 `include` 命令包含在源文件前部。



Note



问题 171 如何进行函数的嵌套调用?



问题阐述

Note

C 语言中不允许进行嵌套的函数定义,因此各函数之间是平行的,不存在上一级函数和下一级函数的问题。但是 C 语言允许在一个函数的定义中出现对另一个函数的调用,这就是函数嵌套调用。那么如何嵌套调用呢?

专家解答

函数的嵌套调用,即在被调函数中又调用其他函数。

下面通过一个分数相加的例子来介绍如何进行函数的嵌套调用。本例要实现的是,在程序运行时,输入 4 个数。前两个数分别是一个分数的分子和分母,后两个数也是一个分数的分子和分母。按 Enter 键以后,将这两个分数相加的计算结果显示出来。

代码如下。

```
#include<stdio.h>
int gys(int x,int y)                /*定义求最大公约数函数*/
{
    return y?gys(y,x%y):x;         /*递归调用 gys(), 利用条件语句返回最大公约数*/
}
int gbs(int x,int y)                /*定义求最小公倍数函数*/
{
    return x/gys(x,y)*y;
}
void yuefen(int fz,int fm)          /*定义约分函数*/
{
    int s=gys(fz,fm);
    fz/=s;
    fm/=s;
    printf("the result is %d/%d\n",fz,fm);
}
void add(int a,int b,int c,int d)   /*定义加法函数*/
{
    int u1,u2,v,fz1,fm1;
    v=gbs(b,d);                     /*调用函数求公倍数*/
    u1=v/b*a;
    u2=v/d*c;
    fz1=u1+u2;
    fm1=v;
    yuefen(fz1,fm1);                /*调用函数进行约分*/
}
main()
```




```

{
    int a,b,c,d;
    scanf("%ld,%ld,%ld,%ld",&a,&b,&c,&d);
    add(a,b,c,d);                /*调用加法函数*/
}

```

程序运行结果如图 10.5 所示。

下面分析一下函数嵌套调用执行的过程：在执行 main() 函数的过程中，当遇到调用 add() 函数的操作语句时，流程转向 add() 函数；在执行 add() 函数时，当遇到调用 gbs() 函数的操作语句时，流程转向 gbs() 函数；在执行 gbs() 函数时，当遇到调用 gys() 函数的操作语句时，流程转向 gys 函数；当完成 gys() 函数的全部操作后返回到 gbs() 中；当完成 gbs() 剩下的全部操作后，返回到 add() 中执行剩余部分；直到 add() 函数结束，再返回到 main() 函数。

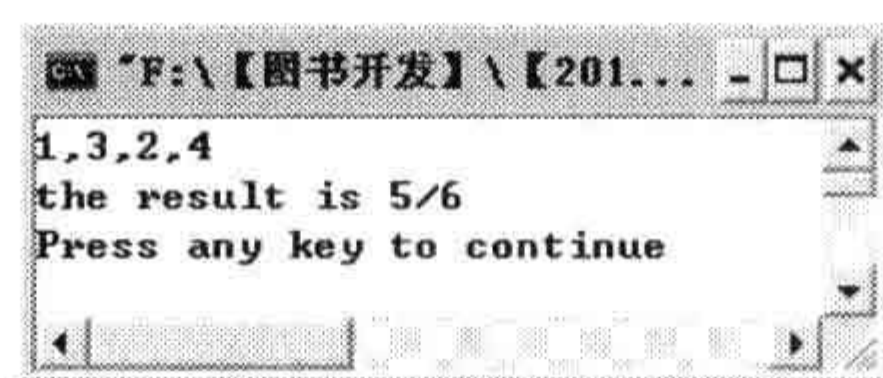


图 10.5 分数相加



Note

专家点评

通过函数的嵌套调用，可以把一个复杂问题分解成多个简单的问题进行处理。

问题 172 什么是递归调用？如何实现？

问题阐述

在学习 C 语言前，也许读者就常常听到递归调用，那么到底什么是递归调用呢？又该如何递归调用呢？

专家解答

在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的递归调用。这种函数称为递归函数。C 语言允许函数的递归调用。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。从上面的描述中会得出这样一个结论：递归调用就是一个无终止的自身调用。在编写程序的过程中，很明显不需要这种无终止的递归调用。这时就需要设置一个条件作为递归调用的出口，使其在此终止，而不会无休止地运行下去。

来看一个简单的递归调用实例。有 5 个人坐在一起，问第五个人多少岁，他说比第四个人大 2 岁。问第四个人的岁数，他说比第三个人大 2 岁。问第三个人，又说比第二个人大 2 岁。问第二个人，说比第一个人大 2 岁。最后问第一个人，他说是 10 岁。编写程序，当输入第几个人时求出其对应年龄。

```

#include<stdio.h>
int age(int n)                /*自定义函数 age()*/
{

```




Note

```

int f;
if(n==1)
    f=10;                                /*当 n 等于 1 时, f 等于 10*/
else
    f=age(n-1)+2;                        /*递归调用 age()函数*/
return f;                                /*将 f 值返回*/
}
main()
{
    int i,j;                             /*定义变量 i,j 为基本整型*/
    printf("Do you want to know whose age?please input:\n");
    scanf("%d",&i);                      /*输入 i 的值*/
    j=age(i);                             /*调用函数 age()求年龄*/
    printf("the age is %d",j);            /*将求出的年龄输出*/
}

```

程序运行结果如图 10.6 所示。

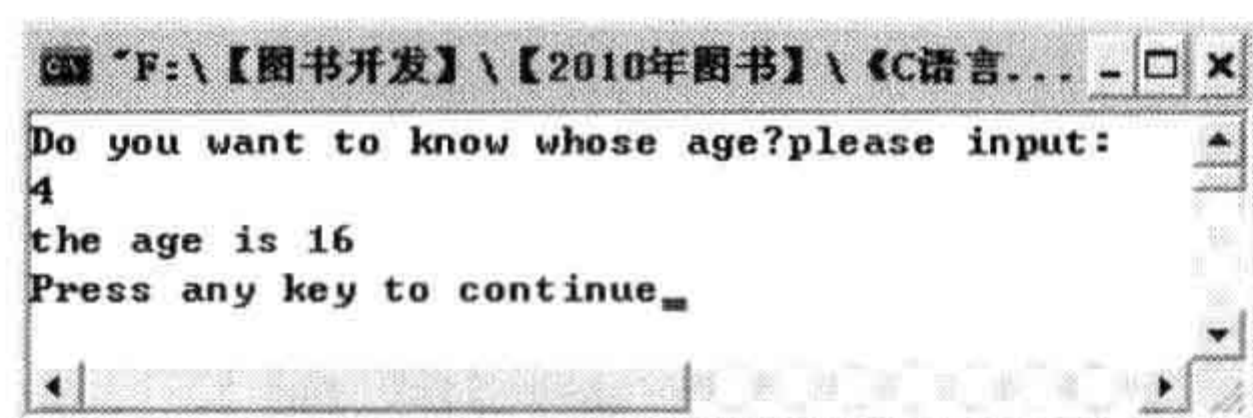


图 10.6 求年龄

递归的过程分为两个阶段：第一阶段是“回推”，如图 10.7 所示。由题可知，要想求第五个人的年龄必须知道第四个人的年龄，要想知道第四个人的年龄必须知道第三个人的年龄……直到第一个人的年龄，这时 $\text{age}(1)$ 的年龄已知，就不用再推。第二阶段是“递推”，如图 10.8 所示。从第一个人推出第二个人，再从第二个人推出第三个人的年龄……一直推到第五个人的年龄为止。这里要注意，必须要有一个结束递归过程的条件。本实例中就是当 $n=1$ 时， $f=10$ ，也就是 $\text{age}(1)=10$ ，否则递归过程会无限制地进行下去。总之，递归就是在调用一个函数的过程中又出现直接或间接地调用该函数本身。

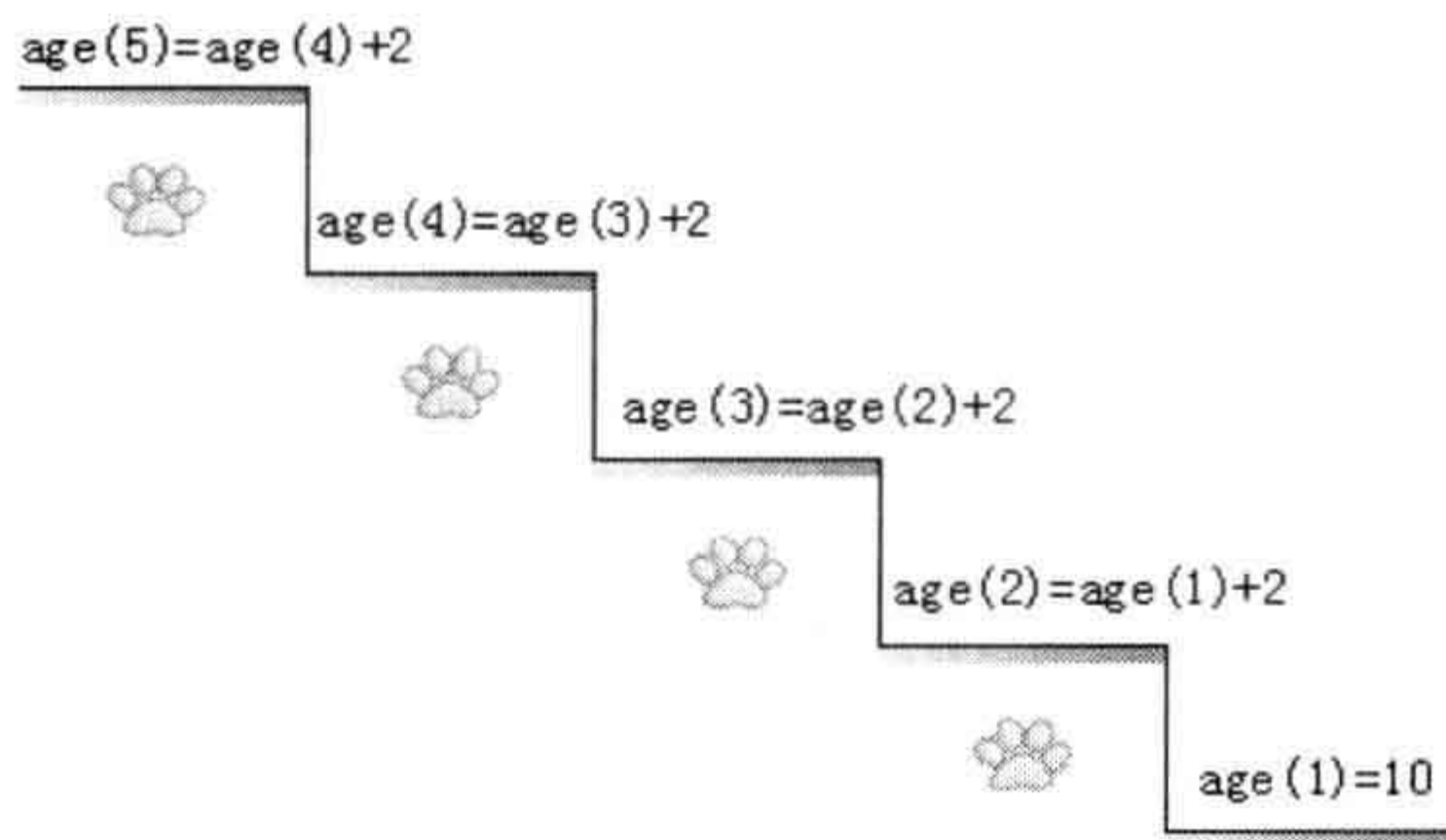


图 10.7 回推

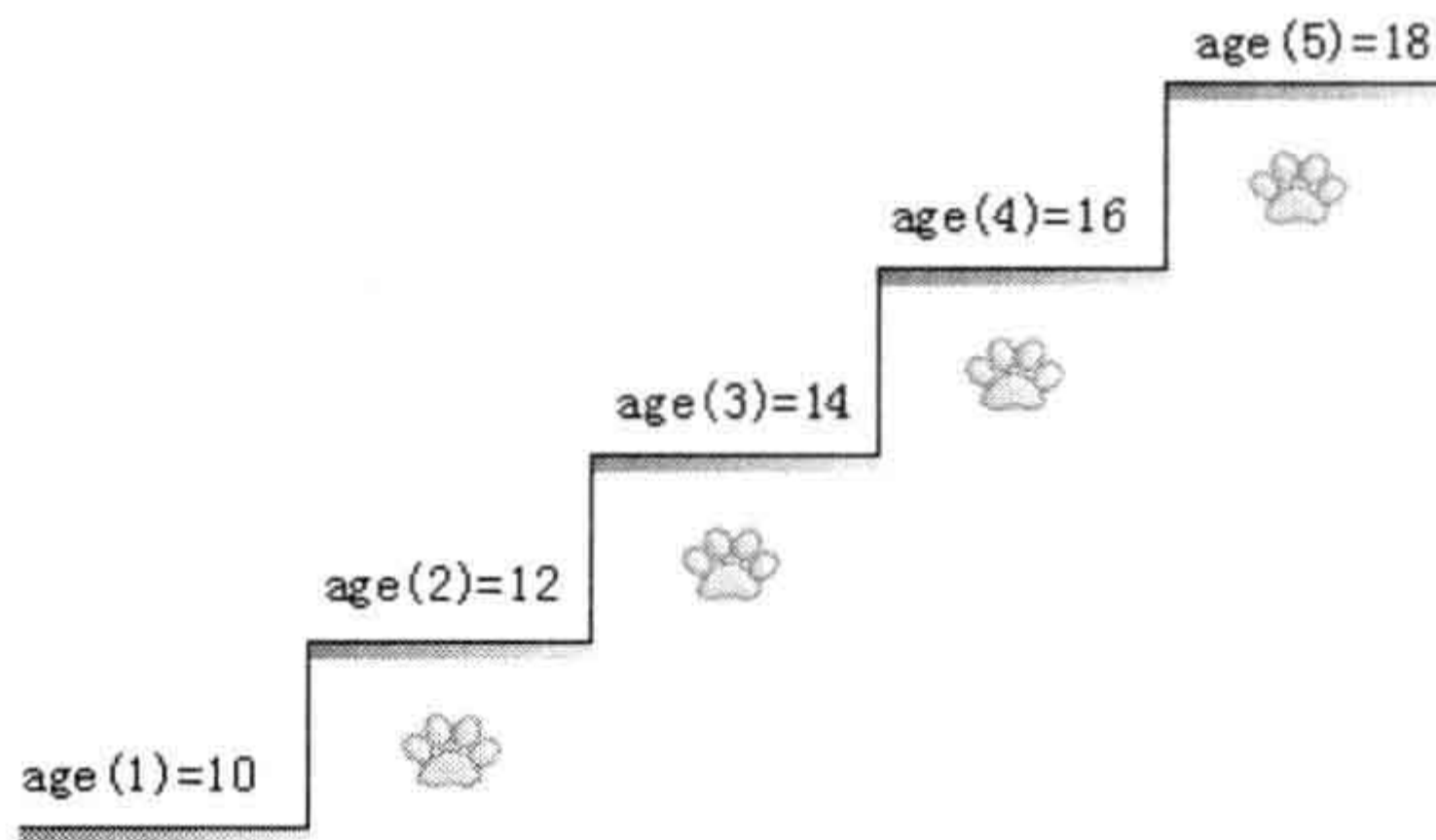


图 10.8 递推

这里，“回推”可以看成是个“下楼梯”的过程，而“递推”则是个“上楼梯”的过程，将这两个结合起来就是递归。

递归的例子很多。下面再来介绍一个简单的例子，就是求一个数的阶乘。



通过键盘输入一个数，求该数的阶乘。代码如下。

```
#include<stdio.h>
main()
{
    int n,value;
    printf("please input n:\n");
    scanf("%d",&n);
    value=factor1(n)                /*调用 factor1()函数*/;
    printf("%d factorial is:%d",n,value);
}
int factor1(int n)                 /*自定义 factor1()函数*/
{
    int result;
    if(n==1)
        return 1;
    result=factor1(n-1)*n;         /*递归调用 factor1()函数*/
    return result;
}
```



Note

程序运行结果如图 10.9 所示。

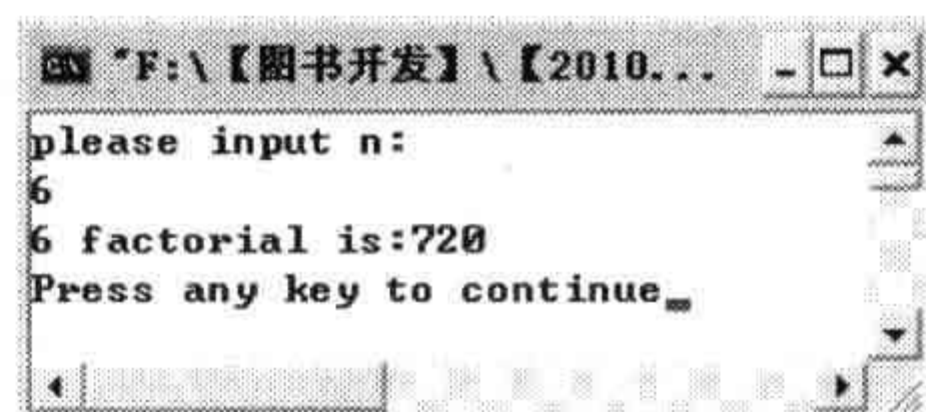


图 10.9 求阶乘

该程序的非递归写法如下。

```
#include<stdio.h>
main()
{
    int n,value;
    printf("please input n:\n");
    scanf("%d",&n);
    value=factor2(n);              /*调用 factor2()函数*/
    printf("%d factorial is:%d",n,value);
}
int factor2(int n)                /*自定义非递归函数 factor2()*/
{
    int i,result;
    result=1;
    for(i=1;i<=n;i++)
        result=result*i;          /*实现求阶乘*/
    return result;
}
```




Note

非递归函数 `factor2()` 的执行比较好理解。这里用到了前面讲过的循环结构，它应用一个从 1 开始到指定数值结束的循环。在循环中，用“变化”的乘积依次去乘每个数。而 `factor1()` 的递归执行比 `factor2()` 略微显得复杂些。当输入的数值为 1，也就是程序中的 `n` 值为 1，调用 `factor1()` 时，函数返回值为 1；除此之外的其他值调用将返回 `factor(n-1)*n` 这个乘积。为了求出这个表达式的值，就需要一层层向下推，调用 `factor()` 一直到 `n` 等于 1，调用开始返回。这就是前面讲过的递推即下楼梯的过程。

当函数调用自己时，在栈中为新的局部变量和参数分配内存，函数的代码用这些变量和参数重新运行。递归调用并不是把函数代码重新复制一遍，仅仅参数是新的。当每次递归调用返回时，原来的局部变量和参数就从栈中消除，从函数内此次函数调用点重新启动运行。可递归的函数被称之为对自身的“推入和拉出”，大部分递归程序没有明显地减少代码规模和节省内存空间。

递归函数的主要优点是可以把算法写得比使用非递归函数时更清晰、更简洁。在没有理解递归如何使用时，往往是一见递归调用就一头雾水，等到真正明白递归后，许多问题就开始喜欢用递归来实现，特别是与人工智能有关的问题，更适宜用递归方法来解决。递归的另一个优点是，递归函数不会受到怀疑，较非递归函数而言，某些人更相信也更愿意使用递归函数。

专家点评

函数的多次递归调用可能会造成堆栈的溢出，但一般情况下是不会发生堆栈溢出现象的，除非一个递归程序运行失去控制。

问题 173 函数如何将数组元素作为实参？

问题阐述

对于函数参数的问题前面已作了讲解，那么如何将数组元素作为函数的实参使用呢？

专家解答

数组元素与普通变量并无区别。数组元素只能用作函数实参，其用法与普通变量完全相同。用数组元素作实参和变量作实参一样，是单向传递的。

例如，通过键盘输入 10 个数据，计算相邻两个数的和。代码如下。

```
#include<stdio.h>
void add(int x,int y)                /*自定义函数 add(), 计算两个数相加的结果*/
{
    int z;
    z=x+y;                          /*计算两个数相加*/
    printf("%5d",z);
}
```




```
main()
{
    int a[10],i;
    printf("please input 10 numbers:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);          /*为一维数组中的元素赋初值*/
    printf("the result is:\n");
    for(i=0;i<9;i++)
    {
        add(a[i],a[i+1]);          /*调用 add()函数*/
    }
    printf("\n");
}
```

程序运行结果如图 10.10 所示。

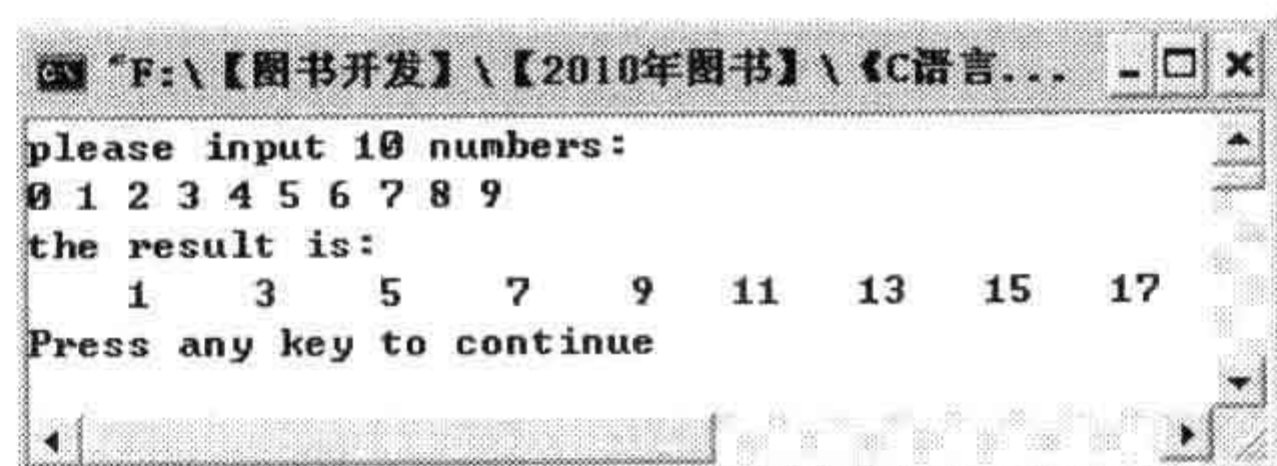


图 10.10 计算相邻两数之和

语句:

```
add(a[i],a[i+1]);
```

就实现了用数组元素作实参。

再来看一个例子，统计字符串中大写字母的个数。代码如下。

```
int cap(char c)          /*自定义 cap()函数，用来判断字母是不是大写字母*/
{
    if (c>='A'&& c<='Z')
        return 1;          /*如果是大写字母，则返回 1*/
    else
        return 0;          /*如果不是大写字母，则返回 0*/
}
main()
{
    int i,num=0;          /*自定义变量，并给 num 赋值为 0*/
    char str[100];          /*自定义字符型数组*/
    printf("Input a string: ");
    gets(str);          /*获取输入的字符串*/
    for(i=0;str[i]!='\0';i++)
        if (cap(str[i]))          /*调用函数判断字母是不是大写字母*/
            num++;
    puts("the string is:");
    puts(str);          /*输出字符串*/
}
```




```
printf("num=%d\n",num);
```

```
/*输出大写字母个数*/
```

```
}
```

程序运行结果如图 10.11 所示。



Note

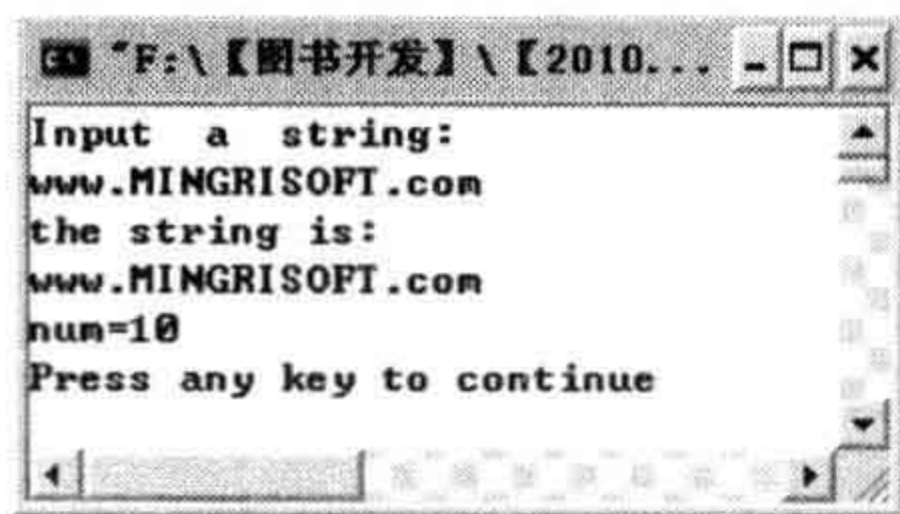


图 10.11 统计大写字母个数

专家点评

用数组元素作实参时，只要数组类型和函数的形参变量的类型一致，那么作为下标变量的数组元素的类型也就和函数形参变量的类型是一致的。因此，并不要求函数的形参也是下标变量。总之，数组元素的处理与普通变量的处理方式一样。用数组名作函数参数时，则要求形参和相对应的实参都必须是类型相同的数组，并且必须有明确的数组声明。当形参和实参二者不一致时，就会发生错误。具体如何使用数组名作参数，将在问题 174 中详细讲解。

问题 174 如何将数组名作为函数参数？

问题阐述

上面讲解了如何使用数组元素作为实参，那么如何将数组名作为函数参数使用呢？

专家解答

在编写程序的过程中，可以用数组名作为函数参数。这种方法实际上是通过数组的首地址传递整个数组。

通过下面的例子来看看如何用数组名来作函数的参数。求学生平均身高，代码如下。

```
#include<stdio.h>
float average(float array[],int n)          /*自定义求平均身高函数*/
{
    int i;
    float aver,sum=0;
    for(i=0;i<n;i++)
        sum+=array[i];                    /*用 for 语句实现 sum 累加求和*/
    aver=sum/n;                             /*总和除以人数求出平均值*/
    return(aver);                          /*返回平均值*/
}
main()
```




Note

```

{
    float height[100],aver;
    int i,n;
    printf("please input the number of students:\n");
    scanf("%d",&n);                /*输入学生数量*/
    printf("please input student`s height:\n");
    for(i=0;i<n;i++)
        scanf("%f",&height[i]);    /*逐个输入学生的身高*/
    printf("\n");
    aver=average(height,n);         /*调用 average()函数求出平均身高*/
    printf("average height is %6.2f\n",aver); /*将平均身高输出*/
}

```

程序运行结果如图 10.12 所示。

前面提到过用数组名作函数参数，实际上就是通过数组的首地址传递数组。这样，两个数组就共占同一段内存单元。这时如果形参数组中各元素的值发生变化，会使实参数组元素的值同时发生变化。

例如，通过键盘任意输入 10 个数据，使用直接插入排序法对这组数字由小到大进行排序。

插入排序是把一个记录插入到已排序的有序序列中去，使整个序列在插入了该记录后仍然有序。插入排序中较简单的一种方法便是直接插入排序，其插入位置的确定是通过将待插入的记录与有序区中的各记录自右向左依次比较其关键字值大小来确定的。

假设输入的一组数据为：25，12，36，45，2，9，39，22，98，37。

原始顺序：25 12 36 45 2 9 39 22 98 37

则直接插入排序的过程如表 10.1 所示。

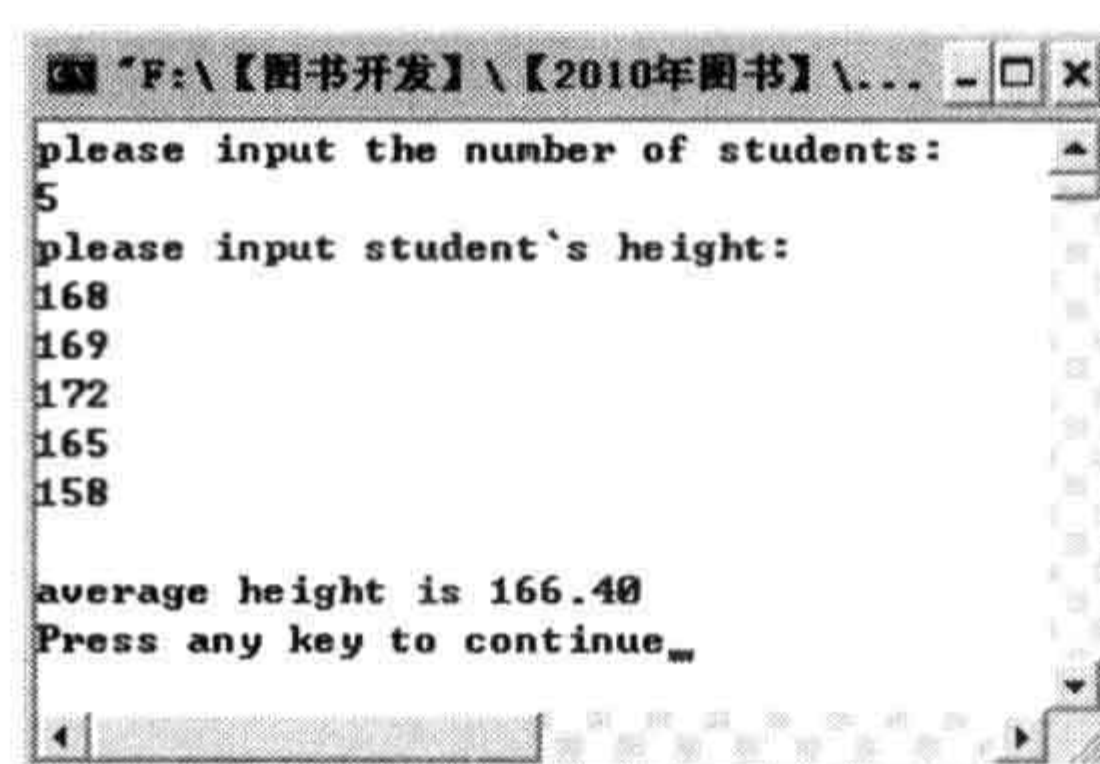


图 10.12 求平均身高

表 10.1 直接插入排序的过程

趟 数	监 视 哨	排 序 结 果
1	12	(12, 25,) 36, 45, 2, 9, 39, 22, 98, 37
2	36	(12, 25, 36,) 45, 2, 9, 39, 22, 98, 37
3	45	(12, 25, 36, 45,) 2, 9, 39, 22, 98, 37
4	2	(2, 12, 25, 36, 45,) 9, 39, 22, 98, 37
5	9	(2, 9, 12, 25, 36, 45,) 39, 22, 98, 37
6	39	(2, 9, 12, 25, 36, 39, 45,) 22, 98, 37
7	22	(2, 9, 12, 22, 25, 36, 39, 45,) 98, 37
8	98	(2, 9, 12, 22, 25, 36, 39, 45, 98,) 37
9	37	(2, 9, 12, 22, 25, 36, 37, 39, 45, 98)

代码如下。

```
#include<stdio.h>
```




```
void insert(int s[], int n)          /*自定义函数 insert()*/
{
    int i, j;
    for (i = 2; i <= n; i++)          /*数组下标从 2 开始, 0 作监视哨, 1 一个数据无可比性*/
    {
        s[0] = s[i];                 /*给监视哨赋值*/
        j = i - 1;                   /*确定要进行比较的元素的最右边位置*/
        while (s[0] < s[j])
        {
            s[j + 1] = s[j];          /*数据右移*/
            j--;                      /*移向左边一个未比较的数*/
        }
        s[j + 1] = s[0];              /*在确定的位置插入 s[i]*/
    }
}

main()
{
    int a[11], i;                    /*定义数组及变量为基本整型*/
    printf("please input number:\n");
    for (i = 1; i <= 10; i++)
        scanf("%d", &a[i]);          /*接收从键盘中输入的 10 个数据到数组 a 中*/
    printf("the original order:\n");
    for (i = 1; i < 11; i++)
        printf("%5d", a[i]);          /*将排序前的数组输出*/
    insert(a, 10);                   /*调用自定义函数 insert()*/
    printf("\nthe sorted numbers:\n");
    for (i = 1; i < 11; i++)
        printf("%5d", a[i]);          /*将排序后的数组输出*/
    printf("\n");
}
```

程序运行结果如图 10.13 所示。

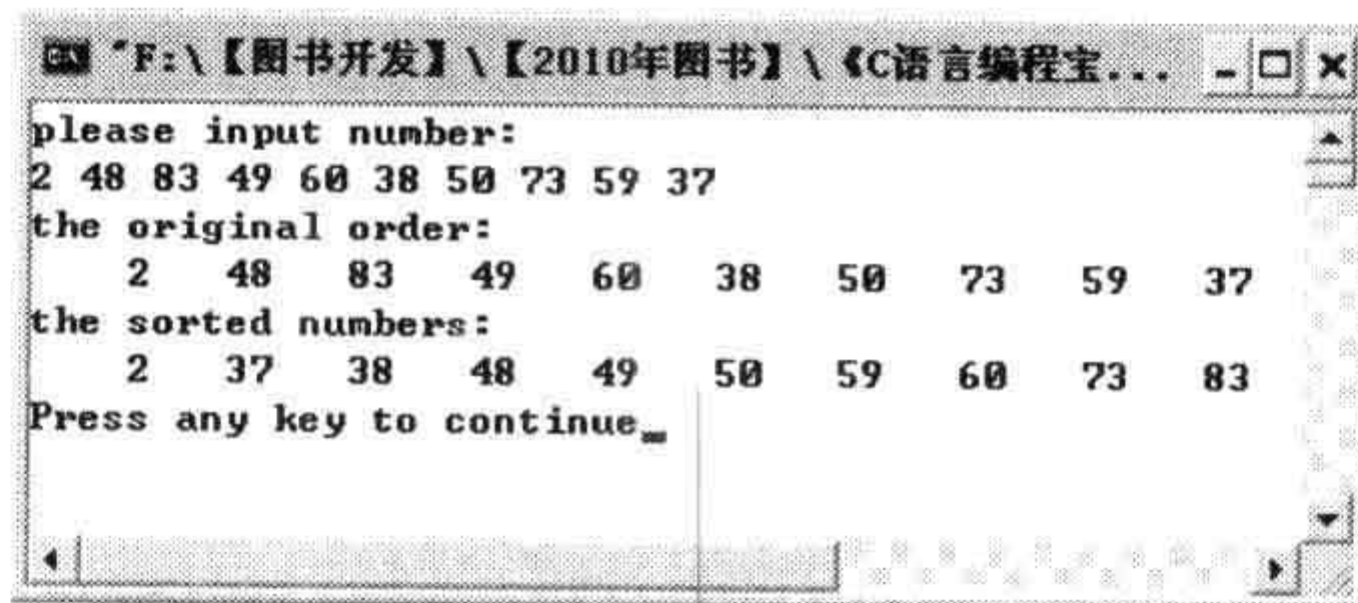


图 10.13 插入排序

用数组名作函数参数与用数组元素作实参有如下几点不同:

- ☑ 用数组元素作实参时, 只要数组类型和函数的形参变量的类型一致, 那么作为下标变量的数组元素的类型也就和函数形参变量的类型是一致的。因此, 并不要求函数的形参也是下标变量。换句话说, 对数组元素的处理是按普通变量对待的。用数组名作函数参数时, 则要求形参和相对应的实参都必须是类型相同的数组, 都必须有明确的数组声明。当形参和实参二者不一致时, 即会发生错误。



- ☑ 在普通变量或下标变量作函数参数时，形参变量和实参变量是由编译系统分配的两个不同的内存单元。在函数调用时发生的值传送是把实参变量的值赋予形参变量。在用数组名作函数参数时，不是进行值的传送，即不是把实参数组的每一个元素的值都赋予形参数组的各个元素。因为实际上形参数组并不存在，编译系统不为形参数组分配内存。那么数据的传送是如何实现的呢？之前曾介绍过，数组名就是数组的首地址。因此，在数组名作函数参数时所进行的传送只是地址的传送，也就是把实参数组的首地址赋予形参数组名。形参数组名取得该首地址之后，也就等于有了实在的数组。实际上是形参数组和实参数组为同一数组，共同拥有一段内存空间。

专家点评

用数组名作为函数参数时，还应注意以下几点。

- ☑ 形参数组和实参数组的类型必须一致，否则将引起错误。
- ☑ 形参数组和实参数组的长度可以不同，因为在调用时，只传送首地址而不检查形参数组的长度。当形参数组的长度与实参数组不一致时，虽不至于出现语法错误（编译能通过），但程序运行结果将与实际不符。这一点应注意。

问题 175 如何将多维数组名作为函数参数？

问题阐述

子函数执行时，整个多维数组是由主函数决定的，这时就要把多维数组的数组名作为函数参数传递给子函数。那么在 C 程序中，怎样将多维数组名作函数参数进行传递？

专家解答

以二维数组为例，其格式如下。

形参定义：

```
fun( Type array[][N])
{
}
```

或

```
fun( Type (*array)[N])
{
}
```

实参定义：

```
main()
{Type a[N];
```




```
...
fun(a)
...
}
```



Note

注意实际参数只写数组名，形式参数可以是数组形式，也可以是指针形式。不管是哪种形式，原二维数组的第一维都可以不声明大小，但其他维则必须声明。多维数组与此类似，即只有第一维可以省略大小。

专家点评

多维数组的数组名作函数的参数，可以由二维数组类推而得出。

问题 176 什么是局部变量？

问题阐述

C 程序中的变量有局部变量和全局变量，那么什么是局部变量呢？

专家解答

局部变量也称为内部变量，即在函数内部定义的变量。它只在本函数范围内有效，在函数外是不能使用该变量的。例如：

```
int f1(int a)
{
    int b,c;
    ...
}
int f2(int x, int y)
{
    int z;
}
main()
{
    int m,n;
}
```

在上述代码中，变量 a、b 和 c 的有效使用范围仅限于 f1() 函数内，变量 x、y 和 z 的有效使用范围在 f2() 函数内，变量 m 和 n 在主函数范围内有效。

在上述代码中再加入一个 f3() 函数，代码如下。

```
int f3(int x)
{
    int y,z,p,q;
```




```
...
}
```

这里，f3()函数中的部分参数和f2()中的部分参数相同。在此要注意的是，虽然它们的名称相同，但是代表的对象不同，之间互不干扰。

复合语句中也可以定义变量，其作用域只在复合语句范围内，这种复合语句也可称为“分程序”或“程序块”。例如：



Note

```
main()
{
    int i,j;
    ...
    {
        int k;
        k=i*j;
        ...
        {
            int l;
            l=k+i+j;
        }
        ...
    }
    ...
}
```

Diagram illustrating variable scope ranges:

- l 的作用范围** (Scope of l): Indicated by a bracket next to the innermost block containing `int l;` and `l=k+i+j;`.
- k 的作用范围** (Scope of k): Indicated by a bracket next to the middle block containing `int k;` and `k=i*j;`.
- i、j 的作用范围** (Scope of i, j): Indicated by a bracket next to the outermost block containing `int i,j;`.

上述程序中定义的 `k` 和 `l` 只在它们所属的复合语句中有效，当离开它们所属的复合语句时，这些变量就无效了。

专家点评

以下事项需要读者注意。

- ☑ 形式参数也同样被看作局部变量，像前面 `f1()` 函数中的形参 `a`、`f2()` 函数中的形参 `x` 和 `y` 等都是局部变量。
- ☑ 主函数中定义的变量也只在主函数中有效，而不因为在主函数中定义了就在整个程序中都有效。当然在主函数中也不能使用其他函数中定义的变量。例如，在上面的程序，在 `mian()` 函数中不能使用 `f1()` 函数中定义的任意变量 (`a`、`b`、`c`)。

问题 177 什么是全局变量？如何应用？

问题阐述

上面讲解了局部变量的相关内容，那么与之相对的全局变量又是什么样的？如何使用呢？

专家解答

全局变量也称为外部变量，即在函数之外定义的变量，其有效作用范围从定义变量的位置开始到本源文件结束。

例如：

```
int a,b;
void f1()
{
    ...
}
float x,y;
int f2()
{
    ...
}
char c1,c2;
char f3()
{
    ...
}
main()
{
    ...
}
```

a、b 的作用范围

x、y 的作用范围

c1、c2 的作用范围

a、b、x、y、c1、c2 都是全局变量，但是它们的作用范围不同，在 main()、f3()、f2() 及 f1() 函数中可以使用变量 a 和 b，在 main()、f3()、f2() 函数中可以使用变量 x 和 y，在 main()、f3() 函数中可以使用变量 c1 和 c2。

来看一个具体的应用实例。通过键盘输入一组数据，找出这组数据中的最大数与最小数，将最大数与最小数的位置互换，将互换后的这组数据再次输出。代码如下。

```
#include<stdio.h>
int min=10000,max=0;
void change(int a[],int n)
{
    int i,j,k;
    for (i = 0; i < n; i++)           /*找出数组中最小的数*/
        if (a[i] < min)
        {
            min = a[i];
            j = i;                     /*将最小数所存储的位置赋给 j*/
        }
    for (i = 0; i < n; i++)           /*找出这组数据中的最大数*/
        if (a[i] > max)
```




```

{
    max = a[i];
    k = i; /*将最大数存储的位置赋给 k*/
}
a[k] = min; /*在最大数位置存放最小数*/
a[j] = max; /*在最小数位置存放最大数*/
printf("\nthe position of min is:%3d\n", j); /*输出原数组中最小数所在的位置*/
printf("the position of max is:%3d\n", k); /*输出原数组中最大数所在的位置*/
printf("Now the array is:\n");
for (i = 0; i < n; i++)
    printf("%5d", a[i]);
}
main()
{
    int a[20], i, n; /*定义数组及变量数据类型为基本整型*/
    printf("please input the number of elements:\n");
    scanf("%d", &n); /*输入要输入的元素个数*/
    printf("please input the element:\n");
    for (i = 0; i < n; i++) /*输入数据*/
        scanf("%d", &a[i]);
    change(a,n);
    printf("\nmax=%5d\nmin=%5d",max,min);
}

```



Note

程序运行结果如图 10.14 所示。

图 10.14 最大数与最小数互换位置

在上述程序中，max、min 是全局变量，所以在函数 change()和 main()中都可以引用。在 main()函数中调用 max 和 min 输出的结果和在 change()函数中调用 max 和 min 的值是一致的。如果此时要在 main()函数中输出 k 的值，则会提示如图 10.15 所示的错误。

图 10.15 提示错误



提示错误是因为 k 是局部变量，其作用范围仅限于 change() 函数中，所以在 main() 函数中调用时会提示未定义 k 这个变量。

专家点评

建议少使用全局变量，其原因有以下 3 点。

- ☑ 全局变量在程序的整个执行过程中都将占用存储单元，并不是在使用时才在内存中为其开辟单元。
- ☑ 使用全局变量会使函数的可移植性降低，因为函数的执行要依赖于其所在的外部变量。如果要将一个函数移到另一个文件中，还要将有关的外部变量及其值一起移过去。如果再出现和其他变量重名的问题，就会更麻烦。
- ☑ 全局变量使用过多会降低程序的清晰性，因为各个函数执行时都有可能改变全局变量的值。

问题 178 存储方式有哪几种？分别是什么？

问题阐述

什么叫存储方式？存储方式有几种？分别是什么？

专家解答

因变量存储方式不同而产生的特性称作变量的生存期。生存期表示了变量存在的时间。生存期加上前面讲过的作用域是从时间和空间这两个不同的角度来描述变量的特性，这两者既有联系，又有区别。

图 10.15 展示了内存中供用户使用的存储空间的情况。

静态存储变量通常是在变量定义时就分配固定的存储单元并一直保持不变，直至整个程序结束。前面讲过的全局变量即属于此类存储方式，它们存放在如图 10.16 所示的静态存储区中。动态存储变量是在程序执行过程中，使用它时才分配存储单元，使用完毕立即将该存储单元释放。例如前面讲过的函数的形式参数，在函数定义时并不给形参分配存储单元，只是在函数被调用时才予以分配，调用函数完毕立即释放。此类变量存放在如图 10.16 所示的动态存储区中。从以上分析可知，静态存储变量是一直存在的，而动态存储变量则时而存在、时而消失。

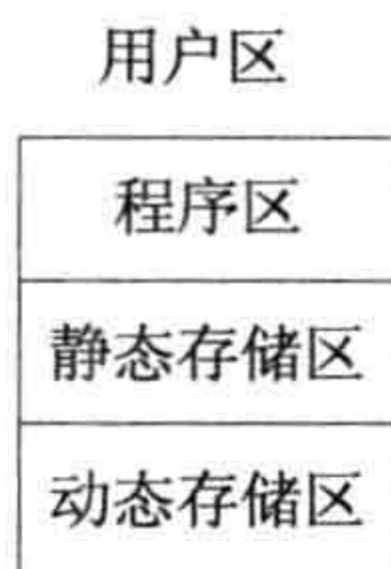


图 10.16 内存中用户区



专家点评

存储方式是指变量使用内存的方式。

问题 179 如何使用 auto 关键字?



Note

问题阐述

这种存储类型是 C 语言程序中使用最广泛的一种类型。C 语言规定,函数内凡未加存储类型声明的变量均视为自动变量,也就是说自动变量可省去声明符 auto。那么要如何使用 auto 关键字呢?

专家解答

在前面章节的程序中所定义的变量,凡未加存储类型声明符的都是自动变量。例如:

```
{  
    int i,j,k;  
    ...  
}
```

等价于:

```
{  
    auto int i,j,k;  
    ...  
}
```

自动变量具有以下特点:

(1) 自动变量的作用域仅限于定义该变量的个体内。在函数中定义的自动变量只在该函数内有效,在复合语句中定义的自动变量只在该复合语句中有效。例如:

```
int f1(int a)  
{  
    auto int x,y;  
    ...  
    {  
        auto char ch;  
        ...  
    }  
    ...  
}
```

ch 的作用范围

x、y 的作用范围

(2) 自动变量属于动态存储方式,只有在使用它,即定义该变量的函数被调用时,才给它分配存储单元,函数调用结束后将之释放存储单元。因此,函数调用结束之后,自动变量的值不能保留。同样,在复合语句中定义的自动变量在退出复合语句后也不能再使



用, 否则将引起错误。

例如, 输入两个数, 如果两数均不为 0, 且前一个数大于后一个数, 则求两数之差, 否则求两数之和。



Note

```
#include<stdio.h>
main()
{
    auto int i,j,k;                /*定义变量为 auto 型*/
    printf("please input the number:\n");
    scanf("%d,%d",&i,&j);          /*输入两个变量, 分别赋予 i 和 j*/
    if(i!=0&&j!=0)
        if(i>j)                    /*判断 i 是否大于 j, 是则相减, 否则相加*/
            k=i-j;
        else
            k=i+j;
    printf("the result is %d\n",k); /*输出 k 的值*/
}
```

程序运行结果如图 10.17 所示。

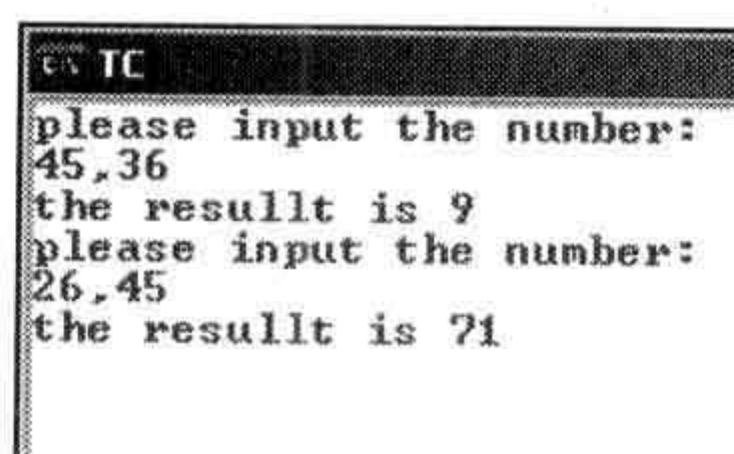


图 10.17 求差或和

若将上面程序改为:

```
#include<stdio.h>
main()
{
    auto int i,j;
    printf("please input the number:\n");
    scanf("%d,%d",&i,&j);          /*输入两个变量, 分别赋予 i 和 j*/
    if(i!=0&&j!=0)
    {
        auto int k;                /*定义 auto 型变量, 作用范围在花括号范围内*/
        if(i>j)
            k=i-j;
        else
            k=i+j;
    }
    printf("the result is %d\n",k);
}
```

运行时提示错误, 如图 10.18 所示。



```

Message
Compiling D:\PRO\615.C:
Warning D:\PRO\615.C 14: 'k' is assigned a value which is never used in funct
Error D:\PRO\615.C 15: Undefined symbol 'k' in function main

```

图 10.18 错误提示

之所以出现如图 10.18 所示错误, 是因为 `k` 是在复合语句内定义的自动变量, 只在该复合语句内有效, 而程序中调用 `printf()` 函数来输出 `k` 的值在复合语句之外, 超出了 `k` 的使用范围, 所以才会产生错误。

(3) 由于自动变量的作用域和生存期都局限于定义它的个体内(函数或复合语句内), 因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量, 也可与该函数内部的复合语句中定义的自动变量同名。

来看一个 `auto` 变量的应用示例。

```

#include<stdio.h>
main()
{
    auto int i,j,k;
    printf("please input the number:\n");
    scanf("%d,%d",&i,&j);           /*输入两个变量, 分别赋予 i 和 j*/
    k=i*i+j*j;
    if(i!=0&&j!=0)
    {
        auto int k;                /*定义 auto 型变量, 作用范围在花括号范围内*/
        if(i>j)
            k=i-j;
        else
            k=i+j;
        printf("result1 is %d\n",k); /*输出 k 的值, 有别于花括号外 k 的值*/
    }
    printf("result2 is %d\n",k);
}

```

程序运行结果如图 10.19 所示。

```

TC
please input the number:
5,6
result1 is 11
result2 is 61

```

图 10.19 auto 变量应用

本程序在 `main()` 函数中和复合语句内两次定义了变量 `k` 为自动变量。输出结果 1 中的 `k` 是在复合语句中定义的, 输出结果 2 中的 `k` 是在主函数中定义的。这两个 `k` 虽然同名, 但作用范围不同, 所以最终输出的结果也不同。

专家点评

函数中的局部变量, 如不专门声明为 `static` 存储类型, 都是动态地分配存储空间, 数



Note



据存储在动态存储区中。函数中的形参和在函数中定义的变量（包括在复合语句中定义的变量），都属此类。在调用该函数时，系统会给它们分配存储空间，在函数调用结束时将自动释放这些存储空间。这类局部变量称为自动变量。自动变量用关键字 `auto` 进行存储类型的声明。



Note

问题 180 什么是静态变量？如何实现？

问题阐述

在编写程序的过程中，对于某些函数的局部变量的值，有时不希望它在函数调用结束后消失，也就是不释放该变量所占用的存储单元；同样，有时在程序设计中也希望某些外部变量只限于被本文件引用。这就需要使用静态变量来实现了，那么什么是静态变量？如何应用呢？

专家解答

通常使用关键字 `static` 对变量进行声明，实现静态变量。

1. 静态局部变量

在局部变量的声明前再加上 `static` 声明符，就构成静态局部变量。

例如：

```
static int a,b;
static float x,y;
static int a[3]={0,1,2};
```

静态局部变量属于静态存储方式，具有以下特点。

- ☑ 静态局部变量在函数内定义，但不像自动变量那样，当调用时就存在，退出函数时就消失。静态局部变量属于静态存储类型，在静态存储区内分配存储单元，在程序整个运行期间都不释放，即静态局部变量始终存在，也就是说其生存期为整个源程序。
- ☑ 静态局部变量的生存期虽然为整个源程序，但是其作用域仍与自动变量相同，即只能在定义该变量的函数内使用该变量。退出该函数后，尽管该变量还继续存在，但不能使用它。再次调用定义它的函数时，它又可继续使用。
- ☑ 对基本类型的静态局部变量若在声明时未赋予初值，则系统自动赋予 0 值；而对自动变量若不赋初值，则其值是不定的。

下面来看一个用静态局部变量来求累加和的例子，代码如下。

```
#include<stdio.h>
int add(int x)                /*自定义求和函数*/
{
    static int n = 0;          /*定义 static 变量*/
```




```

    n = n + x;
    return n;                                /*将所求结果返回*/
}

main()
{
    int i, j, sum;
    printf("please input the number:\n");
    scanf("%d", &i);
    printf("the result is:\n");
    for (j = 1; j <= i; j++)
    {
        sum = add(j);                        /*调用 add()函数*/
        printf("%d: %d\n", j, sum);          /*输出计算结果*/
    }
}

```



Note

程序运行结果如图 10.20 所示。

从上述程序可以看出，静态局部变量 n 是一种生存期为整个源程序的量。每次调用函数 $\text{add}()$ 时，静态局部变量 n 都保存了前次被调用后留下的值。因此，当需要多次调用一个函数且要求在后一次调用时使用前一次调用保留的某些变量的值时，可考虑采用静态局部变量。

如果将函数 $\text{add}()$ 中的 static 改成 auto ，则运行结果如图 10.21 所示。

```

TC
please input the number:
6
the result is:
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21

```

图 10.20 static 变量应用

```

TC
please input the number:
6
the result is:
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6

```

图 10.21 static 改成 auto 后的结果

从图 10.21 所示的运行结果可以看出，自动变量占用动态存储区空间，而不占用静态存储区空间；每次调用后变量 n 的值都释放，每当再调用时 n 的值都从 0 开始。

2. 静态全局变量

在全局变量的变量类型声明之前加上 static ，就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量自然也是静态存储方式。这两者在存储方式上并无不同，其区别主要在于作用域不同。非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态全局变量在各个源文件中都是有效的。例如，在 file1.c 中定义了非静态全局变量 XX ，则在其他的源文件（如 file2.c ）中也可以调用。静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。例如，在 file1.c 中定义了静态全局变量 YY ，则在其他的源文件（如 file2.c ）中则不能调用。



说明:

static 这个声明符在不同的地方所起的作用是不同的。把局部变量改变为静态变量后, 改变的是它的存储方式, 即改变了其生存期; 把全局变量改变为静态变量后改变的是它的作用域, 限制了其使用范围。



Note

专家点评

在开发一个较大程序时, 往往是由若干个人分工完成, 即每个人完成各自负责的模块。每个人可以独立地在其设计的文件中使用相同的外部变量名而互不相干, 前提是在定义的外部变量前面加上 static, 使其成为静态外部变量。这样大大地提高了程序的模块化和通用性。

问题 181 什么是寄存器变量? 如何实现?

问题阐述

通常变量的值是存放在内存中, 当对一个变量频繁读写时, 则需要反复访问内存储器, 从而花费大量的存取时间。为了提高效率, C 语言提供了另一种变量, 即寄存器变量。那么如何实现呢?

专家解答

寄存器变量允许将局部变量的值存放在 CPU 中的寄存器中使用时不需要访问内存, 而直接从寄存器中读写。寄存器变量的声明符是 register。

register 变量的应用示例如下。

```
#include<stdio.h>
main()
{
    register i, s = 0;                /*定义 register 变量*/
    for (i = 1; i <= 100; i++)
        s = s + i;
    printf("s=%d\n", s);
}
```

程序运行结果如图 10.22 所示。



图 10.22 register 变量应用

本程序循环 100 次, 两个变量 i 和 s 都将随着循环被频繁地调用。为了提高程序运行效率, 所以定义这两个变量为寄存器变量。



专家点评

对寄存器变量还要说明以下几点:

- ☑ 寄存器变量属于动态存储方式,凡需要采用静态存储方式的量不能定义为寄存器变量。
- ☑ Turbo C 允许同时定义两个寄存器变量。当定义了过多的寄存器变量时也不必担心,编译程序会自动地将超过限制数目的寄存器变量当作非寄存器变量来处理。



Note

问题 182 如何声明外部变量?

问题阐述

由于 C 语言允许将一个较大的程序分成若干独立模块文件分别编译,如果一个源文件中的函数想引用其他源文件中的变量,那么就要想到如何声明外部变量。

专家解答

外部变量可以用 `extern` 来声明。这就是说, `extern` 变量可以扩展外部变量的作用域。

1. 在多文件的程序中声明外部变量

定义时缺省 `static` 关键字的外部变量,就是非静态外部变量。其他源文件中的函数引用非静态外部变量时,需要在引用函数所在的源文件中进行声明。其语法格式如下。

```
extern 数据类型 外部变量表;
```

注意:

在函数内的 `extern` 变量声明,表示引用本源文件中的外部变量,而函数外(通常在文件开头)的 `extern` 变量声明,表示引用其他文件中的外部变量。

例如,有一个源程序由源文件 `file1.C` 和 `file2.C` 组成。

`file1.C`

```
int x,y; /*外部变量定义*/
char z; /*外部变量定义*/
main()
{
    ...
}
```

`file2.C`

```
extern int x,y; /*外部变量声明*/
extern char z; /*外部变量声明*/
func (int a,b)
{
```




...

在 file1.C 和 file2.C 两个文件中都要使用 x、y、z 3 个变量。在 file1.C 文件中，把 x、y、z 都定义为外部变量；在 file2.C 文件中，用 extern 把 3 个变量声明为外部变量，表示这些变量已在其他文件中定义，并且这些变量的类型和变量名，编译系统将不再为它们分配内存空间。对构造类型的外部变量（如数组等），可以在声明时进行初始化赋值；若不赋初值，则系统自动定义其初值为 0。

2. 在一个文件内声明外部变量

如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件结束处。此时如果想在定义该变量的位置之前调用此变量，则应该在调用之前用关键字 extern 对该变量进行“外部变量声明”。

在下面的示例程序中，将用 extern 声明外部变量，并将声明的外部变量值输出。代码如下。

```
#include<stdio.h>
main()
{
    extern int X,Y;                /*定义变量 X,Y 为 extern 型*/
    printf("this is an example!\n");
    printf("the extern variable is %d,%d",X,Y);
}
int X=96,Y=88;
```

程序运行结果如图 10.23 所示。

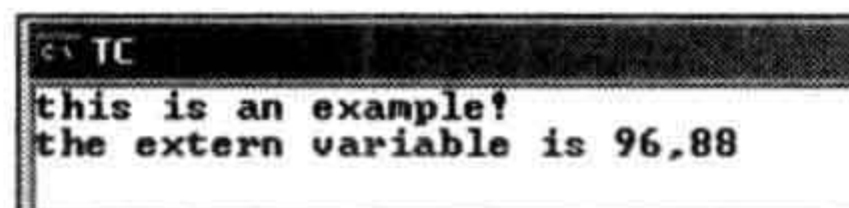


图 10.23 extern 变量应用

专家点评

定义外部变量时，要注意控制的就是它的使用范围，这一点很重要。

问题 183 如何调用编译后的函数？

问题阐述

在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 extern “C”？

专家解答

1. 问题解析

外部函数就是可以被其他源文件调用的函数。定义外部函数时，使用关键字 extern 进



行修饰。在使用一个外部函数时，要先用 `extern` 声明所用的函数是外部函数。

例如，函数头可以写成下面的形式。

```
extern int Add(int iNum1,int iNum2);
```

这样，函数 `Add()` 就可以被其他源文件调用进行加法运算。

2. 答案

C++ 语言支持函数重载，而 C 语言不支持函数重载。函数被 C++ 编译后，在库中的名称与 C 语言的不同。

假设某个函数的原型为：

```
void foo(int x, int y);
```

该函数被 C 编译器编译后，在库中的名称为 `_foo`，而 C++ 编译器则会生成像 `_foo_int_int` 之类的名称。C++ 提供了 C 连接交换指定符号 `extern "C"` 来解决名称匹配问题。

专家点评

C 语言函数不仅仅可以被 C 函数调用，其他语言也可以通过一定的方式来调用 C 语言函数，足可见 C 函数的可用性。

问题 184 如何限定外部变量的使用范围？

问题阐述

在程序设计时，有时需要对某些外部变量限制使用范围，那么该如何实现呢？

专家解答

如果想限制外部变量只应用于本文件，可以加一个 `static` 的声明。例如：

```
static int c;  
void main(){  
....  
....  
}
```

上述代码在一个 `c` 文件中，如果另一个 C 文件使用 `"extern int c"` 这样的语句来调用外部变量 `c` 是不可行的。

在多人开发时，每个人都可以独立的地在其设计的文件中使用相同的外部变量名而互不干扰，只要在外变量名前加一个 `static` 声明就可以了。

专家点评

要注意的是，不要认为对外部变量加了 `static` 声明后，外部变量才是静态存储的，而



Note



不加就是动态存储，其实两种方式下外部变量都是静态存储，只是应用范围不同。

问题 185 如何使用函数调用实现对字符串的统计？

问题阐述

输入一个字符串，要求使用函数调用的方法计算出该字符串共含有多少个字符，如何实现？

专家解答

1. 实现程序前的分析

在此需要自定义一个函数来实现统计字符串中字符个数的功能，最终将统计的个数返回。编写主函数时，在函数体内定义一个数组，使用 `gets()` 函数获得字符串，以数组作实参调用自定义函数，最终将得到的长度值输出。

2. 实现程序

代码如下。

```
#include<stdio.h>
main()
{
    int len;
    char *str[100];
    printf("please input a string:\n");
    /*gets 函数将输入的字符串放入数组 str 中*/
    gets(str);
    len=length(str);
    printf("the string has %d characters.",len);
}
int length(char *p)
{
    int n=0;
    /*当指针未指到字符串结束标志时执行循环体语句*/
    while(*p!='\0')
    {
        n++;
        p++;
    }
    return n;
}
```

/*定义 len 为基本整型变量*/
/*定义字符型指针数组 str*/
/*调用 length() 函数*/
/*将结果输出*/
/*自定义函数 length()*/
/*定义变量 n 为基本整型*/
/*长度加 1*/
/*指针向后移*/
/*返回最终长度*/

专家点评

这是一个综合应用的问题，它不仅考查了函数调用的问题，同时也考查了开发者对字



符串知识的了解。所以说，开发项目是对某类语言的综合应用，开发者要对所用语言有一个总体的理解和掌握。

问题 186 main()函数有什么作用？



Note

问题阐述

main()函数是C语言程序中最重要的一部分，不可或缺，那么它有什么作用？

专家解答

C程序是由一个或多个函数组成的，其中必须有一个且只有一个名为main()的函数，该函数是整个程序的入口。

既然是程序的入口函数，那么在大部分C程序中，main()函数一般都是调用其他函数。例如：

```
int main(){
    welcome();
    menu();
    inprt();
}
```

在C语言中，所有函数的定义都是平等的，也就不存在嵌套定义，但是可以嵌套调用。唯一的例外是，main()函数是不允许其他函数调用的，而且程序从main()函数开始，最后也会回到main()函数结束。

专家点评

main()函数在程序中就是一个开始和结束的引导者，它一般不去实现功能，功能的实现就由其他的子函数和自定义函数来实现。

问题 187 什么是内部函数？

问题阐述

前文中提到了内部函数的概念，那么到底什么是内部函数呢？

专家解答

如果在一个源文件中定义的函数只能被本文件中的函数调用，而不能被同一源程序其他文件中的函数调用，这种函数就称为内部函数。定义内部函数的一般形式如下。

```
static 类型声明符 函数名(形参表);
```




例如:

```
static int f(int a,int b);
```

内部函数也称为静态函数。但此处静态 (static) 的含义已不是指存储方式, 而是指对函数的调用范围只局限于本文件。

使用内部函数的好处是: 不同的人编写不同的函数时, 不用担心自己定义的函数是否会与其他文件中的函数同名, 即使同名也没有关系。

专家点评

内部函数使程序员在编写模块化程序时有了更大的自由, 无须过多考虑函数命名的问题。

问题 188 什么是外部函数? 怎么用?

问题阐述

问题 187 介绍了内部函数, 那么什么又是外部函数呢? 怎么使用?

专家解答

外部函数在整个源程序中都有效, 其定义的一般形式如下。

```
extern 类型声明符 函数名(形参表);
```

例如:

```
extern int f(int a,int b);
```

调用外部函数时, 需要对其进行声明。

```
[extern] 函数类型 函数名(参数类型表)[, 函数名 2(参数类型表 2)...];
```

如在函数定义中没有声明 extern 或 static, 则隐含为 extern。在一个源文件的函数中调用其他源文件中定义的外部函数时, 应用 extern 声明被调用函数为外部函数。

例如:

file1.C

```
main()
{
    extern int fl(int i);           /*外部函数声明, 表示 fl()函数在其他源文件中*/
    ...
}
```

file2.C

```
...
extern int fl(int i);             /*外部函数定义*/
```




```
{  
    ...  
}
```

又如:

f1.C

```
main()  
{  
    extern void input(...),process(...),output(...);  
    input(...);  
    process(...);  
    output(...);  
}
```

f2.C

```
...  
extern void input(...) /*定义外部函数*/  
{  
    ...  
}
```

f3.C

```
...  
extern void process(...) /*定义外部函数*/  
{  
    ...  
}
```

f4.C

```
...  
extern void output(...) /*定义外部函数*/  
{  
    ...  
}
```

专家点评

外部函数是程序模块化的重要实现技术,读者一定要熟练掌握。

问题 189 static()函数与普通函数有什么区别?

问题阐述

前面讲解了一些函数的概念和注意事项,那么 static()函数和普通函数有什么区别呢?



Note



专家解答

1. 答案前的分析

只在当前源文件中使用的函数应该声明为内部函数 (static)，内部函数应该在当前源文件中声明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中声明，要使用这些函数的源文件应包含这个头文件。

使用内部函数的好处是：不同的开发者可以分别编写不同的函数，而不必担心所使用的函数是否会与其他源文件中的函数同名。因为内部函数只能在所在的源文件中使用，所以即使不同源文件有相同的函数名称也没有关系。

2. 参考答案

static()函数与普通函数的区别是：static()函数在内存中只有一份，而普通函数在每个被调用中维持一份备份。

专家点评

这个问题其实就是对一些基本概念的考查，所以只有真正理解了函数的基本概念，才能很好地回答一些概念区别性的问题。

问题 190 形参和实参有什么区别？

问题阐述

形式参数和实际参数都叫参数，那么二者之间的区别是什么？

专家解答

1. 问题分析

本问题答案为书本的理论知识，当然读者也可以通过自身对函数的理解和掌握进行解答。对于这类问题，最终体现在程序中会更加明确。

2. 解析问题

(1) 通过名称理解。

- ☒ 形式参数：按照名称理解，就是形式上存在的参数。
- ☒ 实际参数：按照名称理解，就是实际存在的参数。

(2) 通过作用理解。

- ☒ 形式参数：在定义函数时，函数名后面括号中的变量名称为“形式参数”。在函数调用之前，传递给函数的值将被复制到这些形式参数中。
- ☒ 实际参数：在调用一个函数时，也就是真正使用一个函数时，函数名后面括号中的参数为“实际参数”。函数的调用者提供给函数的参数称为实际参数。实际参数是表达式计算的结果，并且被复制给函数的形式参数。





二者的应用对比如图 10.24 所示。

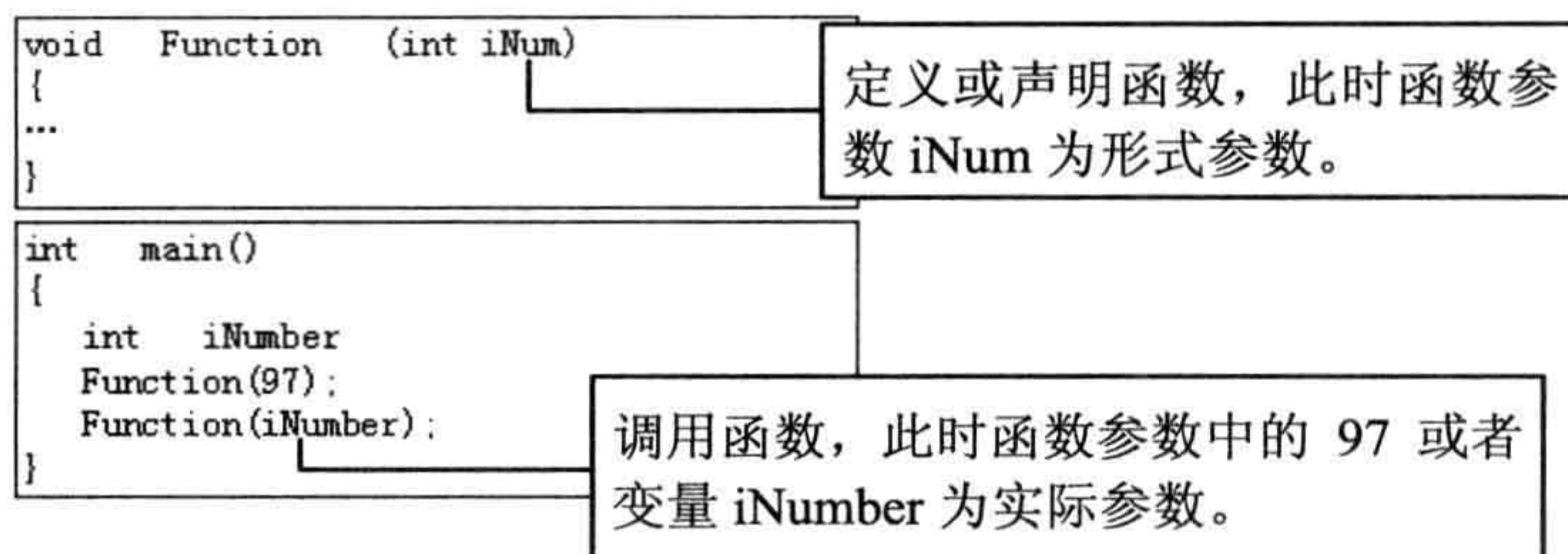


图 10.24 形式参数和实际参数的应用对比

专家点评

函数的一个总结：

使用 C 语言编写的程序，实现的功能都是由各式各样的函数完成的。所以，有时也会把 C 语言称为函数式语言。由此可见，函数是 C 语言源程序的基本模块，是构成 C 程序的基本单元。函数中包含程序的可执行代码，通过对函数模块的调用，即可实现相应的功能。



Note

第 11 章

指针解析

- ▶▶ 什么是指针？什么是指针变量？
- ▶▶ 如何创建指针？
- ▶▶ 如何初始化指针？
- ▶▶ 如何使用指针？
- ▶▶ 函数中如何传递指针？
- ▶▶ 指针、数组和地址之间的关系是什么？
- ▶▶ 如何进行指针运算？
- ▶▶ 如何使用指针操作数组？
- ▶▶ 如何用指针表示多维数组？
- ▶▶ 如何使用指针操作多维数组？
- ▶▶ 如何用指针为函数传递数组？
- ▶▶ 如何用指针表示字符串？
- ▶▶ 如何使用字符串指针作为函数参数？
- ▶▶ 字符数组和字符指针的区别是什么？
- ▶▶ 什么是指针数组？
- ▶▶ 如何使用指针数组处理字符串？
- ▶▶ 如何将指针数组作为函数的参数？
- ▶▶ 什么是指向指针的指针？
- ▶▶ 二级指针如何应用于一维数组？
- ▶▶ 如何实现二级指针对二维数组的操作？
- ▶▶ 二级指针如何操作字符串数组（指针数组）？
- ▶▶ 如何理解返回指针的函数？
- ▶▶ 什么是指向函数的指针？
- ▶▶ 如何用 const 控制指针？
- ▶▶ 什么是“野指针”？
- ▶▶ main() 函数的指针数组形参是怎么回事？
- ▶▶ void 指针就是空指针吗？它有什么作用？
- ▶▶ 指针是一种特殊的变量，只能用来保存地址。这句话对吗？
- ▶▶ 字符指针、浮点数指针以及函数指针这三种类型的变量哪个占用的内存最大？为什么？
- ▶▶ 一个 32 位的机器，该机器的指针是多少位？



问题 191 什么是指针？什么是指针变量？

问题阐述

指针是 C 语言中的一个重要概念，也是 C 语言中的一个重要特色。它的身影在整个 C 语言体系中都会出现，而且其概念也十分复杂，需要多加注意和思考。

专家解答

为了更好地弄清指针的概念，这里不得不先提到地址以及数据在内存中的存储和读取方式。如果在程序中定义了一个变量，在对程序进行编译的时候，系统就会给这个变量分配内存单元，但是这个单元的大小根据定义变量的类型不同而不同。衡量内存单元大小的单位是字节，而内存中的每个字节都有一个编号，这个编号就是地址。类似于现实生活中的旅馆，字节就相当于每个房间，字节编号就相当于旅馆的房间编号，那么客人就相当于存取的数据。对于内存地址的模拟如图 11.1 所示。

图 11.1 中的 1000、1002 等就是内存单元的地址，而 0、1 就是内存单元的内容。换种说法就是基本整型变量 *i* 在内存中的地址从 1000 开始，因为基本整型占两个字节，所以变量 *j* 在内存中的起始地址从 1002 开始，变量 *i* 的内容是 0。

那么指针又是什么呢？这里仅将指针看作是内存中的一个地址。多数情况下，这个地址是内存中另一个变量的位置。如图 11.2 所示。

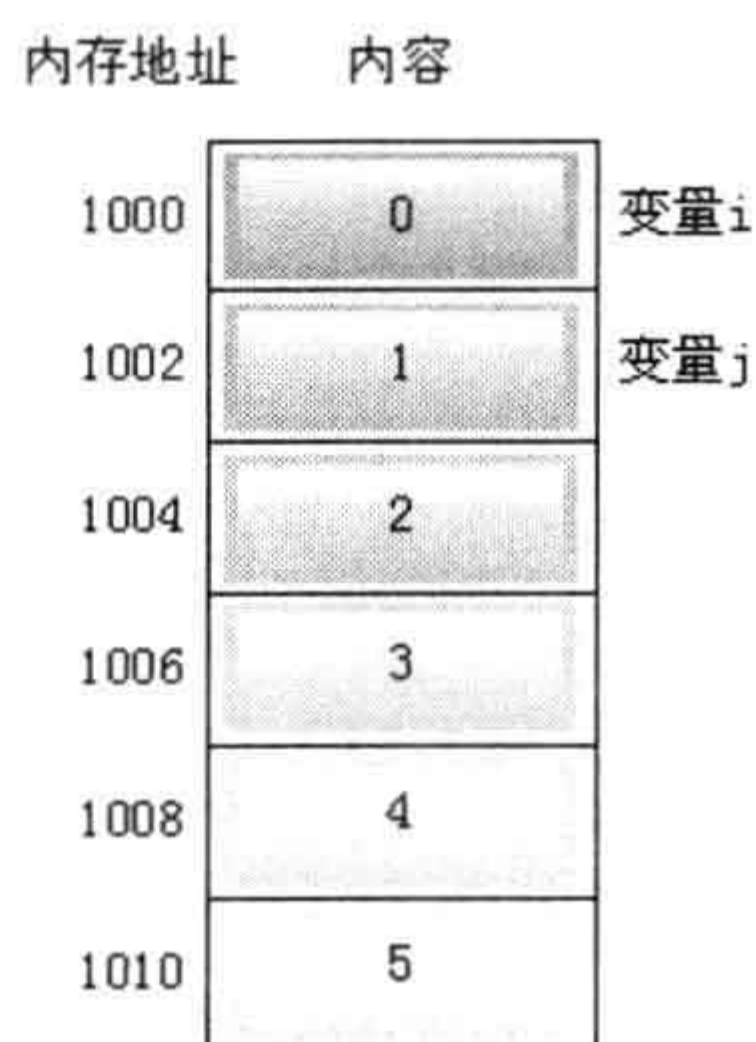


图 11.1 内存地址模拟图

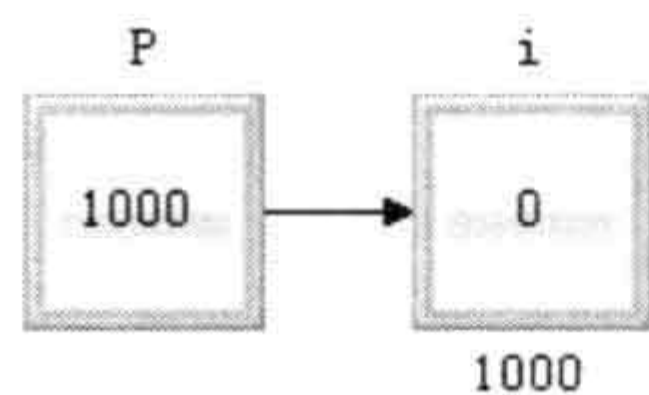


图 11.2 指针指向变量地址

在程序中定义了一个变量，进行编译时就会给这个变量在内存中分配一个地址，通过访问这个地址可以找到所需的变量，这个变量的地址称为该变量的“指针”。如图 11.2 所示，地址 1000 是变量 *i* 的指针。

如果一个变量包含了另一个变量的地址，那么，第 1 个变量就可以说成是指向第 2 个变量。所谓“指向”就是通过地址来体现的。因为指针变量是指向一个变量的地址，所以将一个变量的地址值赋给这个指针变量后，这个指针变量就“指向”了该变量。例如，将变量 *i* 的地址存放到指针变量 *p* 中，*p* 就指向 *i*。其关系如图 11.3 所示。





如图 11.4 所示,在地址 2000 上的这个变量指向地址 2005 上的那个变量,在地址 2000 上这个变量的内容的值是 2005。同理,在地址 2001 上的这个变量指向地址 2004 上的那个变量,在地址 2001 上这个变量的内容的值是 2004。



Note

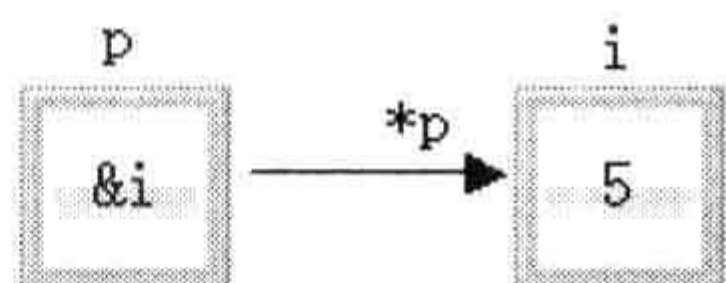


图 11.3 地址与指针

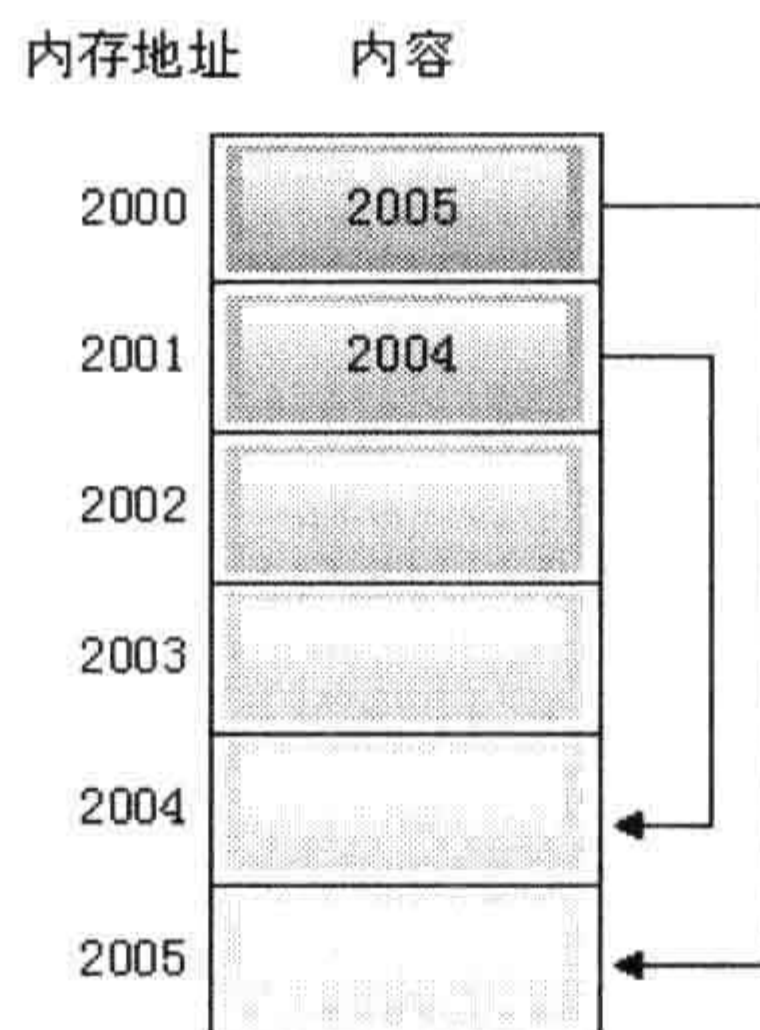


图 11.4 指针变量存放地址

专家点评

能够正确而灵活地运用 C 语言指针是一个成熟 C 语言程序员的标志之一。C 语言指针可以有效地表示复杂的数据结构,能够动态地分配内存,方便字符串的使用,更好地操作数组,完成获取多个函数运行结果的任务。更重要的是,可以直接处理内存单元地址等。总之,学好指针对 C 语言程序员很重要。

问题 192 如何创建指针?

问题阐述

上面的问题使我们了解了什么叫指针,也知道了什么叫指针变量,自然就要发出如何创建指针的疑问。

专家解答

对于指针的创建,并不是一个简单的问题。对于 32 位系统的计算机来说,内存单元的地址要用 4 个字节来表示(16 位系统的是 2 个字节)。也就是说,保存一个指针需要 4 个内存单元。

需要注意的是,如果将内存单元的地址保存到一个 int 型的变量中,那么将不能使用“间接访问”的方式去访问该地址所对应单元中的值。例如:

```
int i,j;
j=&i;
```

上面这两句代码,编译后就是在变量 j 中保存变量 i 的内存地址。但是,这里变量 j 中也仅是把 i 的地址当做一个整数,而并不是一个指针,并不能通过变量 j 去访问变量 i



的内容。上述代码编译时,有的编译器会给出警告,有的甚至不让编译通过。

那么,到底该如何保存指针呢?这就要说到一个特殊的变量,那就是指针变量。对于指针变量的定义格式一般如下:

类型说明 * 变量名;

其中,“*”表示这是一个指针变量,变量名就是定义的指针变量的名称,数据类型说明符就是表示所定义的指针所指向的变量的数据类型。下面就给出一些相应类型的指针变量的示例:

```
int *p;  
float *p;  
char *p;
```

第一行代码定义的是一个整型的指针变量,变量名是 p,也就是说在变量 p 中存着一个内存单元地址,这个地址中保存的是一个 int 型的数据。第二行代码定义的是一个单精度的指针变量,变量名是 p,同样在变量 p 中存着一个内存单元地址,这个地址中保存的是一个 float 型的数据。第三行代码定义的是一个字符型的指针变量,变量名还是 p,变量 p 中还存着一个内存单元地址,这个地址中保存的是一个字符型的数据。

专家点评

这里有个问题必须注意,那就是一个指针变量只能指向同类型的变量,也就是整型的指针变量只能指向整型变量,而不能指向字符变量。为什么呢?这在指针运算的时候就会体现出来,不同类型的指针在运算时移动的字节数是不一样的。

问题 193 如何初始化指针?

问题阐述

明白了什么叫指针,也明白了如何定义指针,下面自然是对指针初始化的问题了,只有初始化的指针才可以使用,这个与普通变量没有区别。

专家解答

定义指针变量之后,必须为其赋具体的值,而且指针变量的赋值只能赋予地址,绝对不可以是其他数据,并且要注意数据类型的一致。下面来看一个问题程序。

这个程序主要是实现对数据之间使用指针进行传递,代码如下。

```
#include<stdio.h>  
#include<stdlib.h>  
int main()  
{  
    int i=10,j;
```



Note



Note

```
float *p;
p=&i;
j=*p;
printf("i=%d\tj=%d\n",i,j);
system("PAUSE");
return 0;
}
```

此程序的运行结果如图 11.5 所示。

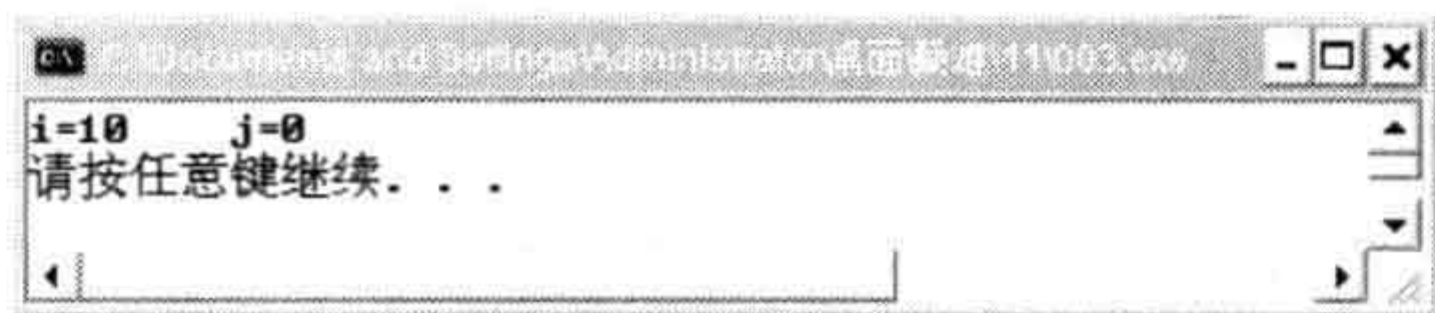


图 11.5 问题程序运行结果

通过图 11.5 的运行结果可以看出，这并不是程序设计之初的结果。为什么会这样呢？

在程序中定义 `int` 类型变量 `i` 和 `j`，并给变量 `i` 赋初值 10，在下一行定义了一个类型为 `float` 的指针变量 `*p`，将 `int` 类型变量 `i` 的地址赋值给 `float` 型的指针变量 `*p`。然后使用间接寻址运算符 `(*)` 从指针变量 `p` 所指向的内存地址中的值进行读取，并将读取的值赋给 `j`，也就是最终设想将 `i` 的值赋给 `j`。

可是，结果却不是当初设想的那样。因为 `p` 指针变量是一个 `float` 类型的指针变量，编译器认为其指向的内存地址保存的是 `float` 类型数据，进而按照 `float` 型数据的格式从内存中读取数据，然后赋给 `j`，所以读出的并不是 `int` 类型的 `i`。

通过上面的问题解析就不难修正问题程序了，只要将“`float *p;`”这一句代码修改为“`int *p;`”就可以了。

专家点评

如果对未初始化的指针进行操作，可能导致系统混乱，甚至导致系统的崩溃，为什么会这样呢？因为刚定义的指针变量中可能有一个随机的值。在指针变量中，该值表示的是一个内存地址。如果该地址正好是操作系统的代码区域，修改该内存中的值就会导致系统崩溃。

问题 194 如何使用指针？

问题阐述

指针变量在初始化以后就可以使用和参与操作了，那么就要用到对指针变量最常用的两个操作符——`*`和`&`。

专家解答

这里又要提到始终贯穿着指针的一个符号“`*`”，但是这里的“`*`”与开始介绍的是不



一样的。开始那个只是指针的说明符，而这里是作为指针运算符使用的，叫做取内容运算符。另一个要介绍的符号在前面的输入输出函数的 `scanf()` 函数中就有使用，那就是 `&`。

上面说明了两种指针变量运算的符号，下面来看一下两者之间的关系，如图 11.6 所示。

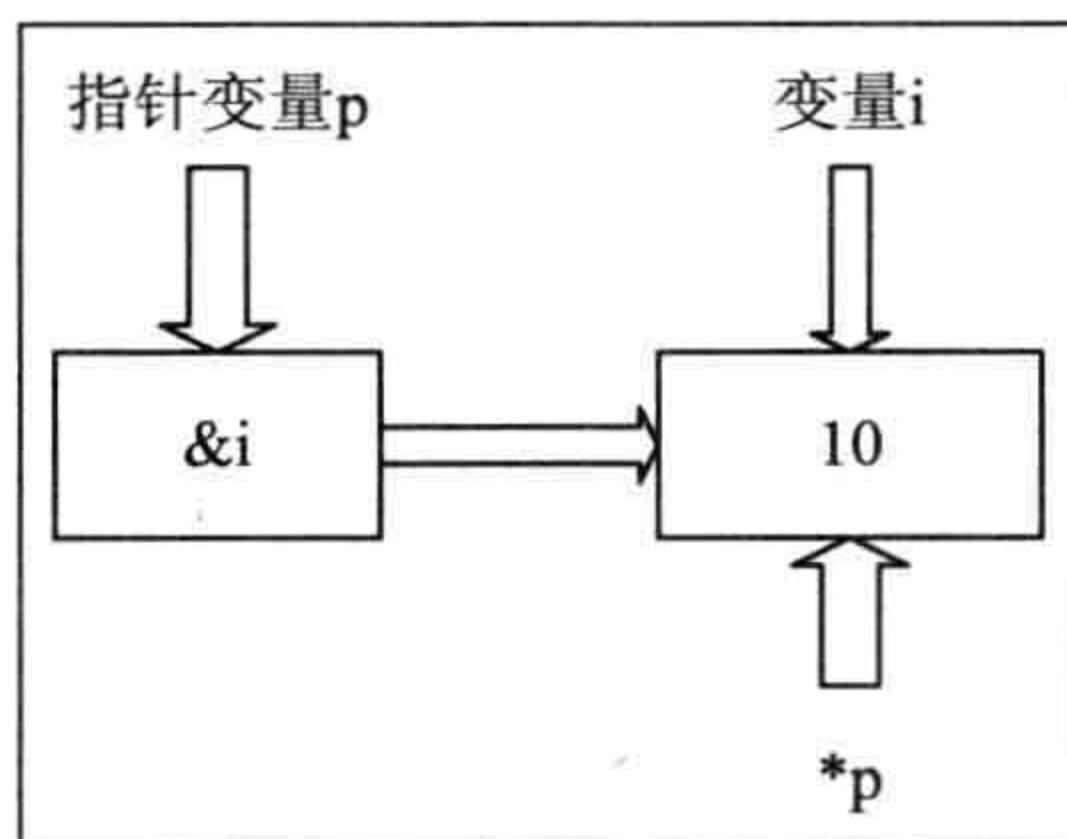


图 11.6 指针与变量之间的关系

通过图 11.6 就不难理解，使用 `&` 符号对变量 `i` 的地址进行读取，读取后存入指针变量 `p` 中。在指针变量 `p` 中保存的是变量 `i` 的地址，那么我们就可以使用 `*p` 对变量 `i` 中的数据进行读取。

下面是一个演示取地址运算符和取内容运算符的程序，代码如下：

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;    int *p;
    p=&i; i=10;
    printf("i=%d\t*p=%d\n",i,*p);
    *p=5;
    printf("i=%d\t*p=%d\n",i,*p);
    system("PAUSE");
    return 0;
}
```

运行结果如图 11.7 所示。

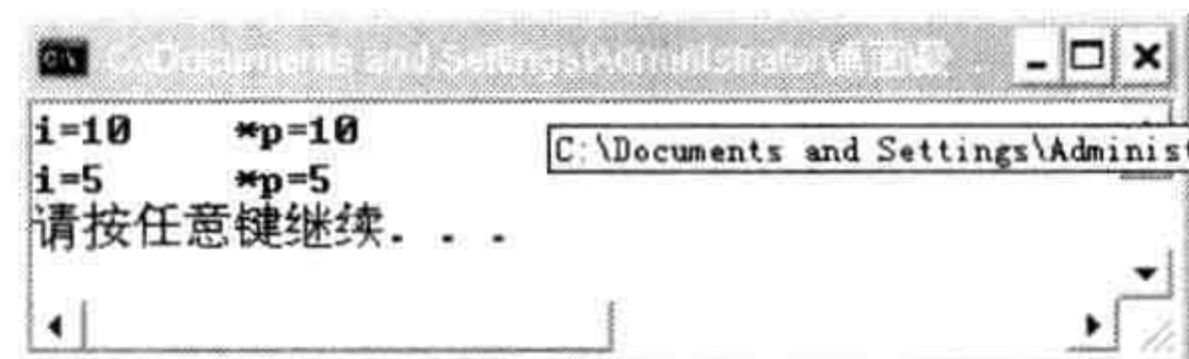


图 11.7 指针变量演示结果

专家点评

上面的演示代码可以这么理解：将 `*p` 看成是变量 `i` 的一个别名，也就是 `i` 和 `*p` 这两个名称都可以访问同一内存单元。





问题 195 函数中如何传递指针？

问题阐述

上文对指针的概念、定义、初始化以及基本的使用方法进行了相应的解答。下面来看一下如何在函数中传递指针。

专家解答

在定义函数时，可以使用指针作为函数的形参。请看如下代码。

```
fs(int *a,int *b){  
    int t;  
    t=*a;  
    *a=*b;  
    *b=t;  
}
```

对于上面这样一个使用指针变量作为形参的函数，在调用的时候就要注意了，因为指针变量保存的是内存地址，因此在调用函数 `fs()` 时，对参数的传入就要传入地址，比如要传入变量 `i` 和 `j`，那么就要写成如下代码进行调用。

```
fs(&i,&j);
```

参数的传递过程如图 11.8 所示。

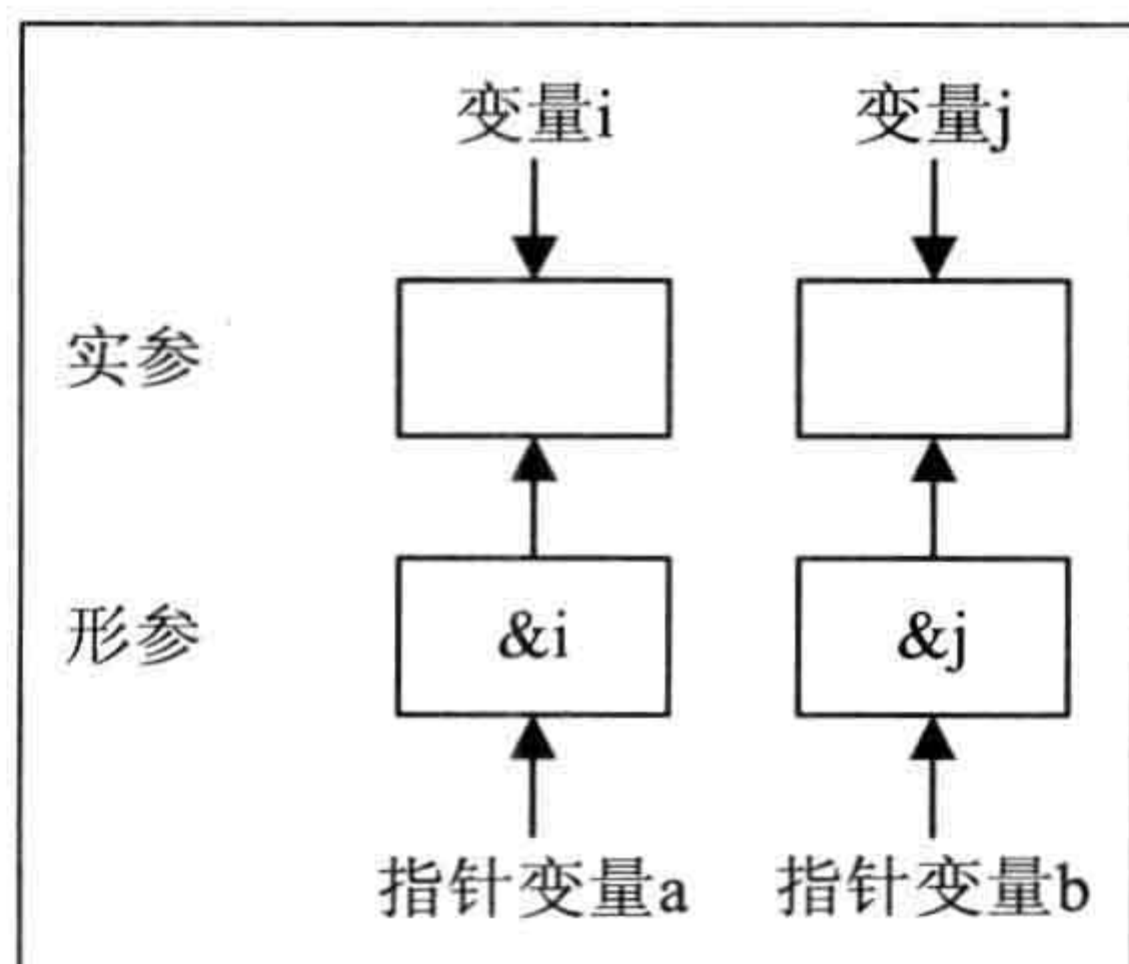


图 11.8 函数参数传递过程

从图 11.8 不难看出，变量 `i` 和指针 `a`，变量 `j` 和指针 `b` 分别是指向同一内存地址的，所以在访问变量的时候，既可以使用变量名，也可以使用指针变量，它们都可以访问同一内容。

专家点评

值得注意的是，在为函数传递字符型参数时，通常都是选择使用指针将字符串的首地



址传递到函数中。

问题 196 指针、数组和地址之间的关系是什么？

问题阐述

每一项事物都不是独立存在的，它与别的事物多少都有些联系。这里讲到了指针，再联想到前面讲到的数组以及本章开头说到的地址，它们之间到底有什么样的关系呢？

专家解答

前面的章节讲到一个数组是保存在一片连续内存单元当中的，而数组名就是这片连续内存单元的首地址，内存单元的地址就是指针，因此数组名也是一个指针。

数组是由多个数组元素组成的，元素按其数组类型的不同，所占连续内存的大小也不相同。一个数组的元素的首地址就是其所占连续内存单元的首地址。

指针变量既可以指向一个数组，也可以指向一个数组元素。将数组名或数组的第一个元素的地址赋给指针，指针就指向了一个数组。如果使指针变量指向第 i 个元素，就可以把 i 元素的首地址赋给它。

请看如下代码。

```
int a[3],b[4];
int *p,*q;
p=a;
q=&b[2];
```

执行以上语句，这段代码的结果如图 11.9 所示。

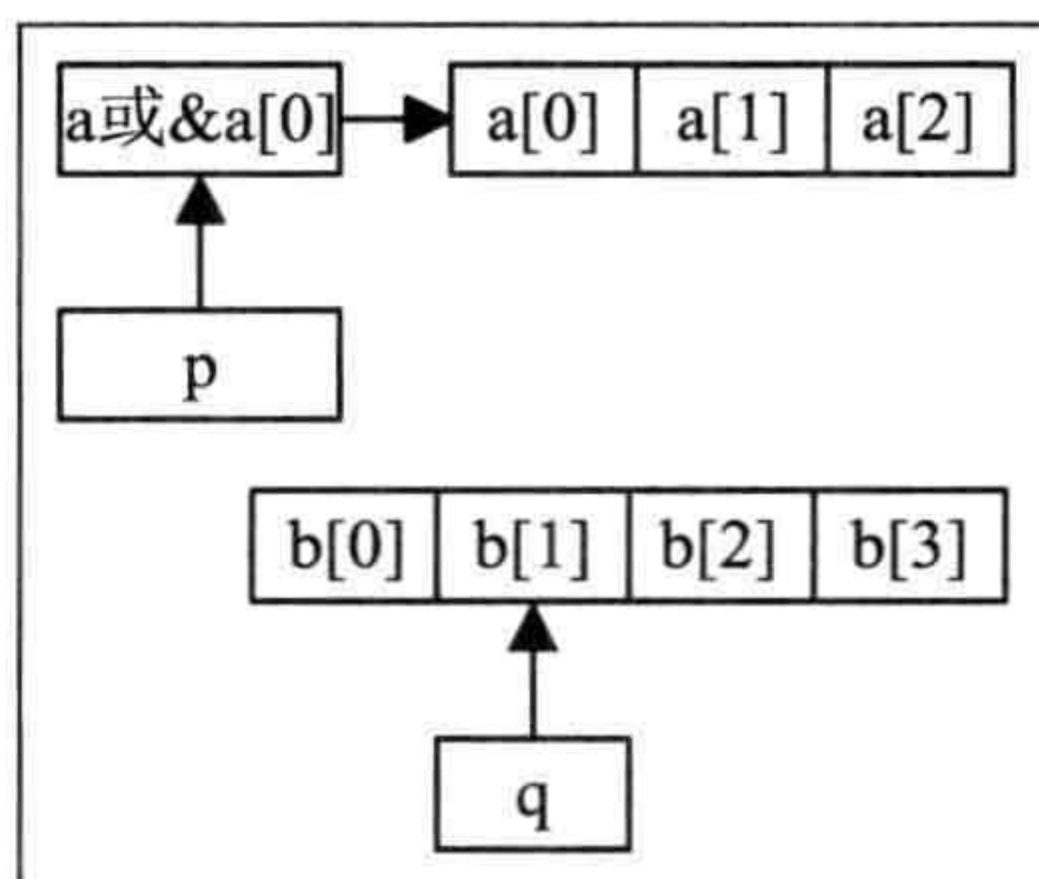


图 11.9 指针、数组和地址的关系

下面是上面代码嵌入程序的的应用。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a[3]={19,89,21},b[4]={19,89,2,1};
```



Note



Note

```

int *p,*q;
p=a;
q=&b[2];
printf("指针变量 p 的值是: %d\n",p);
printf("数组 a 的首地址的值是: %d\n",a);
printf("数组 b 第 3 个元素是: %d\n",b[2]);
printf("指向数组第 3 个元素的指针取出的值是: %d\n\n",*q);
system("PAUSE");
return 0;
}

```

运行结果如图 11.10 所示。

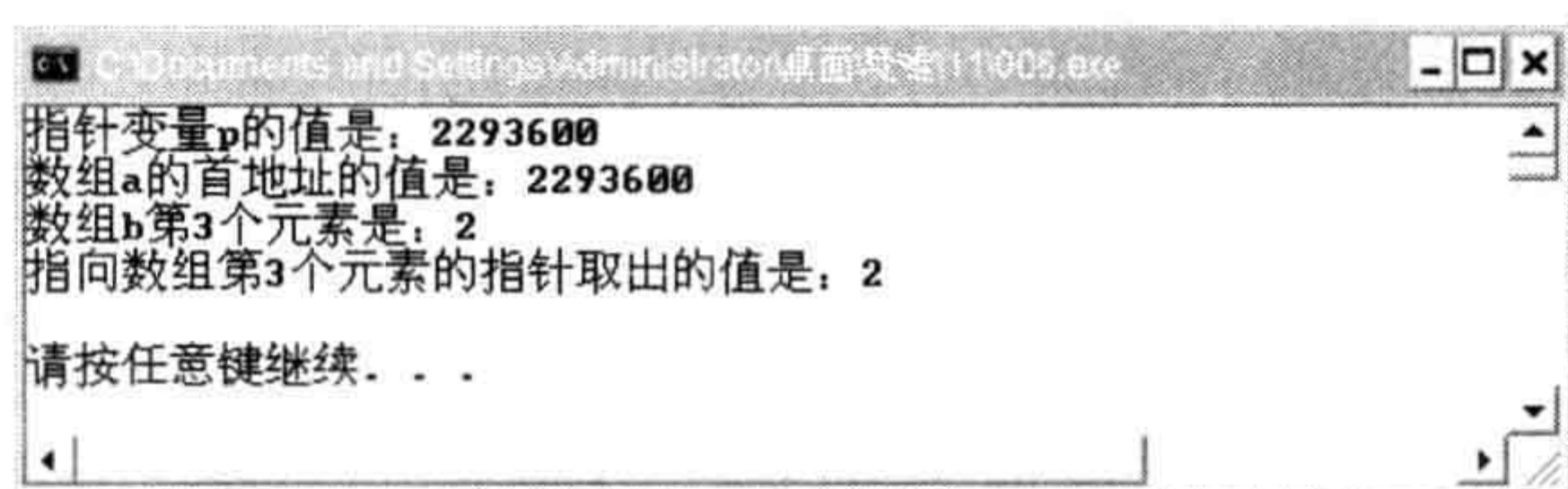


图 11.10 指针、数组和地址关系演示程序运行结果

从程序的运行结果图可以看出，数组名 `a` 和指针变量 `p` 指向的是同一内存地址，用指针变量 `q` 也取出了与使用数组元素标记一样的数值。这也就证实了上面对指针、数组和地址的关系的分析是正确的。

专家点评

使用指针访问数组的元素，可以大大方便对数组元素进行比较、查找等操作。

问题 197 如何进行指针运算？

问题阐述

普通变量可以运算，那么指针可以吗？答案是肯定的。那么如何运算呢，下面就来介绍一下。

专家解答

上一问题中，对指针、数组和地址的关系进行了介绍，最后提到可以利用指针方便地对数组元素进行比较和查找，那么这就需要对指针进行运算。

(1) 自增/自减运算。对于指针变量可以自增/自减运算，例如：

```
p++;
```

这样一句代码究竟有何意义呢？如果 `p` 是一个普通变量，那么它会根据类型不同而自



增一个单位。那么指针变量呢，也是自增一个单位吗？这个单位又会是多少？请看如下代码。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a=0,*p;
    char b='a',*q;
    p=&a;
    q=&b;
    printf("指针 p 指向的地址值是: %d\n",p);
    printf("指针 q 指向的地址值是: %d\n",q);
    p++;
    q++;
    printf("指针自增一次后的地址值是: %d\n",p);
    printf("指针自增一次后的地址值是: %d\n",q);
    system("PAUSE");
    return 0;
}
```

运行结果如图 11.11 所示。

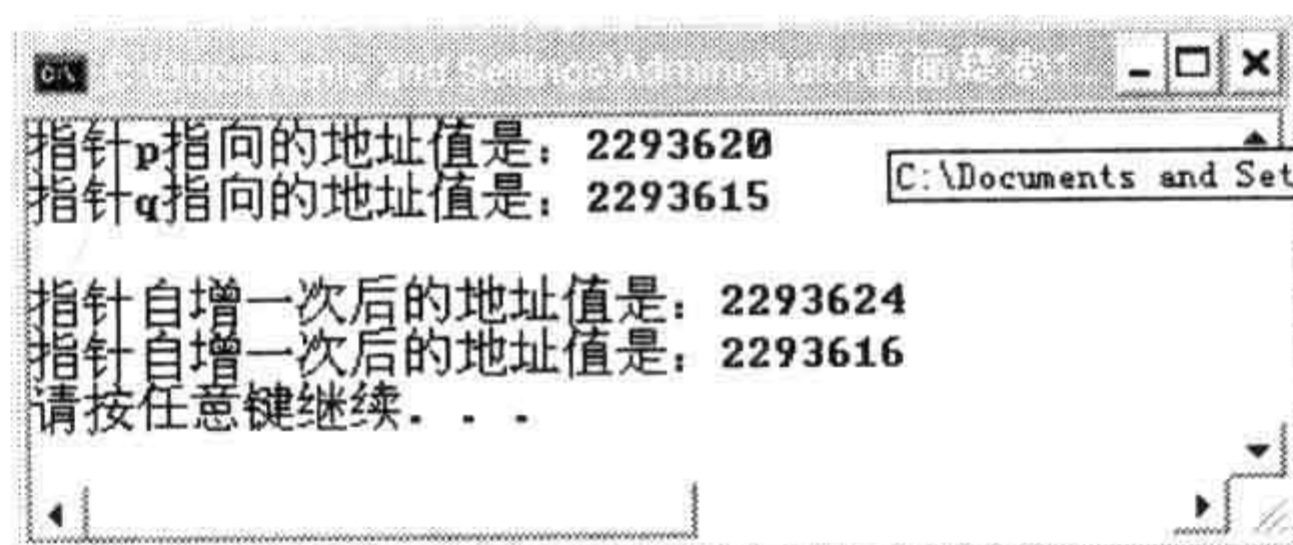


图 11.11 指针自增/自减演示结果

从图 11.11 中可以看出，int 型的指针在自增一次后，其地址值增加了 4，而 char 型指针在自增一次后，其地址值增加了 1。我们知道，C 语言中 int 型数据占 4 个字节的内存单元，而 char 型占 1 个字节的内存单元。这就不难得出，指针的自增和自减一次，就是指针指向的地址值自增或自减相对应数据类型所占内存单元的字节数。

这也就是前面提到为什么说在定义指针变量时，指针的类型非常重要。若使用类型不匹配的指针变量，将得到不可预测的结果。

(2) 指针加减一个整数。指针还可以加减一个整数，那么如何操作呢？例如（int 型的指针 p，p 初始指向地址为 10000）：

```
p+=3;
```

与指针自增/自减相似，这时指针变量 p 并不是将内存单元增加 3 个字节，而是增加 3 个基类数据的字节宽度，也就是相当如下的语句。

```
p=p+3*sizeof(*p)
```



Note



即: $10000+3*4=100012$

指针减去一个整数与增加类似。

(3) 两个指针变量相减。指针变量保存的是内存地址, 所以将两个指针变量相加是没有意义的。于是, 不允许将两个指针变量直接相加, 但是允许两个指针相减。那么两个指针相减得到什么呢? 其实, 通过上面的内容已经不难得出答案, 那就是两个指针相减得到的是两个指针之间相差的内存单元数, 如果再除以与指针类型相对的数据类型宽度, 就会得到两个变量相隔多少个数据类型位置。如果是数据, 就可以得到两个元素之间相隔几个元素。

其实对于指针的加减一个整数和两个指针变量相减, 这些对于与它们相关的普通变量意义不大, 这些主要用于后面要说的指针操作数组。

(4) 指针比较。指针比较是为了判断两个指针在内存中的关系。例如:

$p1 > p2$

如果返回值为 true, 则表示 p1 在内存中处于高位地址。

$p1 == p2$

如果返回值为 true, 则说明这两个指针指向同一地址值。

还有一个就是判断地址值是否为空, 即:

$p == \text{NULL}$

如果返回值为 true, 则说明 p 指针为空。

专家点评

对于指针运算, 要注意的是指针只可以加减一个整数, 其他任何算术运算都是非法的。这一点必须谨记, 否则会造成不可预料的错误和后果。

问题 198 如何使用指针操作数组?

问题阐述

在介绍指针的基本概念和相关运算的问题时, 多次提到数组的问题, 我们心中难免就要问: 指针到底该如何操作数组? 下面就来研究一下这个问题。

专家解答

实践和程序结果可以让我们产生疑问, 但同时也是解决问题的重要依据, 所以首先看一个演示程序。代码如下。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
```




```
int i,a[6]={19,89,2,1,12,25};
int *p=a;
for(i=0;i<=5;i++){
    printf("(a+%d)=%d\n",i,*(a+i));
}
printf("\n");
for(i=0;i<=5;i++){
    printf("*(p+%d)=%d\n",i,*(p+i));
}
system("PAUSE");
return 0;
}
```

运行结果如图 11.12 所示。

从图 11.12 中可以看出, 数组元素 $a[0]$ 和指针 $*(p+0)$, $a[1]$ 和 $*(p+1)$它们每组取出的值都分别相同, 也就说明在引用指针后, 对数组的操作可以使用指针, 而且可以更方便地进行元素的访问。

当使用指针指向数组的首地址后, 指针可以像数组的别名一样, 进行下标操作, 但是这些似乎意义都不大, 指针操作数组的优点主要体现在对数组元素进行顺序操作的时候。

在上面的程序中有这样一段代码。

```
for(i=0;i<=5;i++){
    printf("*(p+%d)=%d\n",i,*(p+i));
}
```

上面这段代码完全可以写成如下的格式。

```
for(i=0;i<=5;i++){
    printf("*p++=%d\n",*p++);
}
```

两段代码中, 第二段就是每执行一次循环, 指针先执行一次取值运算, 然后自加一次, 将指针位置后移一个单位, 一直循环操作, 直到循环体结束。要注意的是, 循环结束后, 指针的位置是在数组所占内存单元的后面一个内存单元, 而不是指向数组的最后一个内存单元。

上面代码中还有这样一段代码。

```
for(i=0;i<=5;i++){
    printf("(a+%d)=%d\n",i,*(a+i));
}
```

那么, 这段代码可不可以改成如下形式呢?

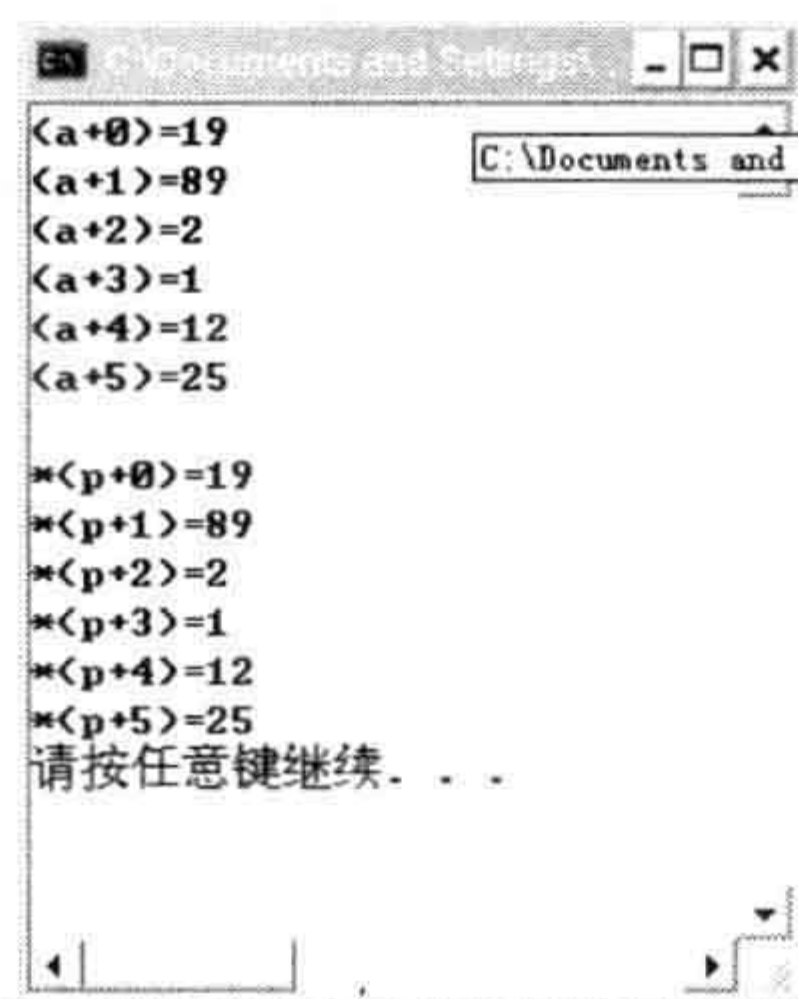


图 11.12 对数组元素的不同方式输出



```
for(i=0;i<=5;i++){  
    printf("*a++=%d\n",*a++);  
}
```

很遗憾，编译出错。这样看来，数组名和指针还是有区别的。那么为什么会出错呢？原因在于指针是指针变量，可以通过运算改变它们的值，但是数组名是常量，常量的值是不允许改变的。

专家点评

操作计算中的几个区分问题：

*p++，由于++是后置运算，这里先得到的是 p 指向变量的值，即先执行*p，然后执行++运算。

*(++p)，这里先运算++p，也就是 p 先自增一次，然后进行取值运算。

(*p)++，先将 p 指针指向的地址值中的数据取出，然后将取出的数据值自增一次运算。

问题 199 如何用指针表示多维数组？

问题阐述

细心的读者会发现，前面只是介绍了一些指针变量指向一维数组和对一维数组元素的操作，而没有提到对二维或多维数组的操作。下面就来看一下如何用指针表示多维数组。

专家解答

这里就是以二维数组为例进行多维数组的操作演示。

首先定义一个二维数组 `int a[3][3]`，数组名代表的是数组的起始地址，因此数组名 a 和第一个元素 `a[0][0]` 的地址是相同的，但是意义却是不同的。

二维数组在逻辑上有一个行列之分，这里我们使用如下一个程序来看一下每一行的头尾元素的地址情况。代码如下。

```
#include<stdio.h>  
#include<stdlib.h>  
int main()  
{  
    int a[3][3];  
    printf("元素 a[0][0]和 a[0][2]的地址分别是: %d,%d\n",&a[0][0],&a[0][2]);  
    printf("元素 a[1][0]和 a[1][2]的地址分别是: %d,%d\n",&a[1][0],&a[1][2]);  
    printf("元素 a[2][0]和 a[2][2]的地址分别是: %d,%d\n",&a[2][0],&a[2][2]);  
    system("PAUSE");  
    return 0;  
}
```




运行结果如图 11.13 所示。

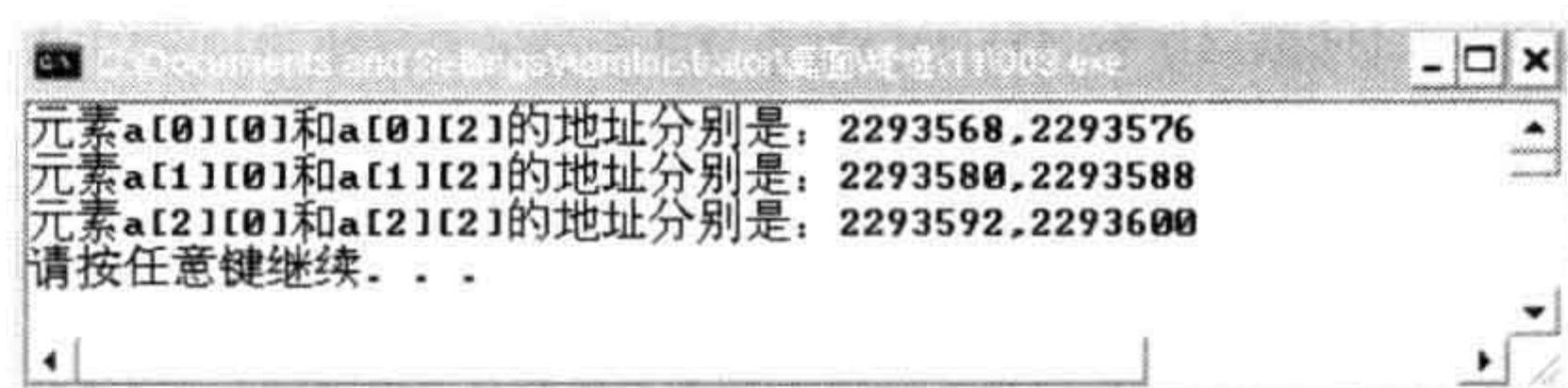


图 11.13 二维数组的地址值

从上图可以看出, 整个二维数组是一个连续的内存单元, 二维数组的数组名也是整个数组的首地址。因此不难得出, 使用指针可以指向二维数组的首地址。

通过上面的程序结果, 可以将二维数组看成一个一维数组, 使用指针操作二维或多维数组的方式与一维数组是类似的。

专家点评

读者在学习的时候要注意, 很多知识点都是相通的, 只要牢固地掌握基本的知识点, 就可以实现对其他扩展知识点的了解。就像习武一样, 再高级精华的招式都是由基本招式组成的, 变化的只是顺序和组合。

问题 200 如何使用指针操作多维数组?

问题阐述

上面问题介绍了如何使用指针表示多维数组, 那么又如何使用指针操作多维数组呢?

专家解答

上一问题以二维数组为例进行讲解, 本问题同样以二维数组为例。

从二维数组的角度来看, a 是二维数组名, a 代表整个二维数组的首地址, 也是二维数组 0 行的首地址, 等于 1000。 $a+1$ 代表第一行的首地址, 等于 1008。如图 11.14 所示。

$a[0]$ 是第一个一维数组的数组名和首地址, 因此也为 1000。 $*(a+0)$ 或 $*a$ 是与 $a[0]$ 等效的, 它表示一维数组 $a[0]$ 0 号元素的首地址, 也为 1000。 $\&a[0][0]$ 是二维数组 a 的 0 行 0 列元素首地址, 同样是 1000。因此, a , $a[0]$, $*(a+0)$, $*a$, $\&a[0][0]$ 是相等的。

同理, $a+1$ 是二维数组 1 行的首地址, 等于 1008。 $a[1]$ 是第二个一维数组的数组名和首地址, 因此也为 1008。 $\&a[1][0]$ 是二维数组 a 的 1 行 0 列元素地址, 也是 1008。因此 $a+1$, $a[1]$, $*(a+1)$, $\&a[1][0]$ 是等同的。

由此可得出: $a+i$, $a[i]$, $*(a+i)$, $\&a[i][0]$ 是等同的。

此外, $\&a[i]$ 和 $a[i]$ 也是等同的。因为在二维数组中不能把 $\&a[i]$ 理解为元素 $a[i]$ 的地址, 不存在元素 $a[i]$ 。C 语言规定, 它是一种地址计算方法, 表示数组 a 第 i 行首地址。由此可以得出: $a[i]$, $\&a[i]$, $*(a+i)$ 和 $a+i$ 也都是等同的。



Note



另外, $a[0]$ 也可以看成是 $a[0]+0$, 是一维数组 $a[0]$ 的 0 号元素的首地址, 而 $a[0]+1$ 则是 $a[0]$ 的 1 号元素首地址, 由此可得出 $a[i]+j$ 是一维数组 $a[i]$ 的 j 号元素首地址, 它等于 $\&a[i][j]$ 。如图 11.15 所示。



Note

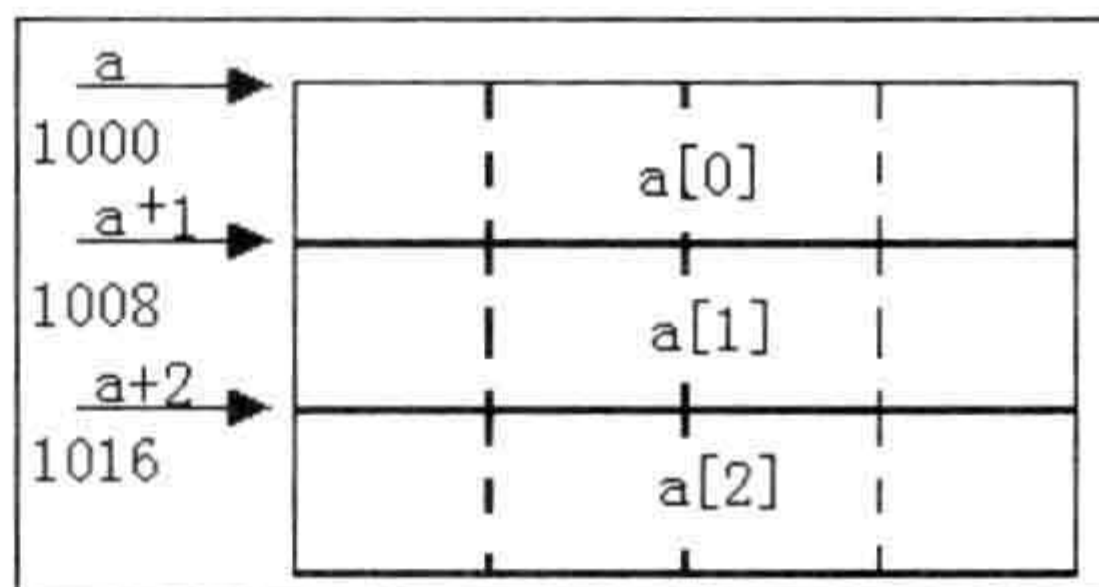


图 11.14 二维数组的行表示

	$a[0]$	$a[0]+1$	$a[0]+2$	$a[0]+3$
a	1000 0	1002 1	1004 2	1006 3
$a+1$	1008 4	1010 5	1012 6	1014 7
$a+2$	1016 8	1018 9	1020 11	1022 12

图 11.15 二维数组的具体表示

由 $a[i]=*(a+i)$ 得 $a[i]+j=*(a+i)+j$ 。由于 $*(a+i)+j$ 是二维数组 a 的 i 行 j 列元素的首地址, 所以, 该元素的值等于 $*(*(a+i)+j)$ 。

指针操作二维数组的基本操作实例程序如下。

```
main(){
    int a[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
    printf("%d",a);
    printf("%d",*a);
    printf("%d",a[0]);
    printf("%d",&a[0]);
    printf("%d\n",&a[0][0]);
    printf("%d",a+1);
    printf("%d",*(a+1));
    printf("%d",a[1]);
    printf("%d",&a[1]);
    printf("%d\n",&a[1][0]);
    printf("%d",a+2);
    printf("%d",*(a+2));
    printf("%d",a[2]);
    printf("%d",&a[2]);
    printf("%d\n",&a[2][0]);
    printf("%d",a[1]+1);
    printf("%d\n",*(a+1)+1);
    printf("%d,%d\n",*(a[1]+1),*(*(a+1)+1));
}
```

专家点评

这里仅是一级指针对二维数组的操作, 但是读者一定要透彻地进行理解, 因为在后面讲到二级指针操作二维数组时要以此为基础。



问题 201 如何用指针为函数传递数组?

问题阐述

上一章节中,在讲解函数问题的时候说到为函数传递数组参数的问题,这里使用指针就会更方便,那么如何用指针为函数传递数组呢?

专家解答

通过前面的章节,我们可以知道数组名就是数组的首地址,当实参向形参传递数组名时,实际上就是传递数组的地址,当形参得到该地址后,就与主调函数中的实参指向同一数组。那么在被调函数中对数组元素的值进行修改后,返回给主调函数。因为我们知道 C 函数只能使用 `return` 语句返回一个值的问题,那么当需要返回多个值时,就可以使用数组作函数的形参。下面来看一个计算二维数组左对角线元素和的应用实例,代码如下。

```
#include<stdio.h>
#include<stdlib.h>
#define M 5
void re(int (*a)[M],int n){
    int i,j;
    for(i=0;i<n;i++){
        printf("请输入数组的第%d 行的%d 个数据: ",i+1,n);
        for(j=0;j<n;j++){
            scanf("%d",&a[i][j]);
        }
        a++;
    }
}
int sum(int (*a)[M],int n){
    int i,s=0;
    for(i=0;i<n;i++){
        s+=a[i][i];
    }
    return s;
}
int main()
{
    int a[M][M];
    re(a,M);
    printf("\n\n");
    printf("二维数组的左对角线元素的和是: %d\n\n",sum(a,M));
    system("PAUSE");
}
```



Note



```
return 0;
```

```
}
```

上面程序的运行结果如图 11.16 所示。

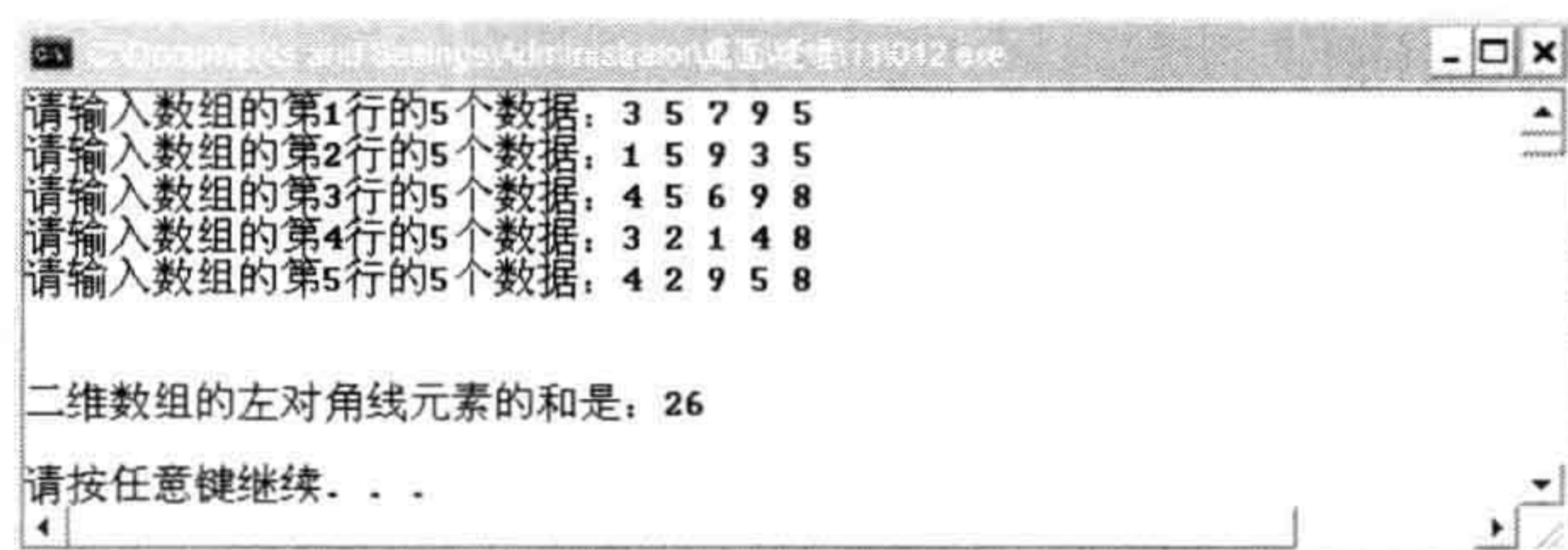


图 11.16 二维数组的左对角线元素和

从上面的程序中不难发现，在 `re()` 函数中使用数组指针的形式 `(*a)[M]`，这样就可以接收二维数组的地址作为实参。在调用的时候，使指针指向实参数组，这样指针就是数组指针，那么语句 `a++` 就是将指针指向下一行，也就是说 `*a` 取的是行首地址，然后 `*a+j` 就是指向具体每行的第 `j` 个元素。

而在 `sum` 求和函数中，最后返回的 `s` 就是数组的左对角线的和。

专家点评

上面程序中只算出了左对角线的和，那么如果想左右都算出呢？这样就可以定义一个一维两个元素的数组，然后将计算出的数值存入这个一维数组，最后返回数组首地址就可以了。

问题 202 如何用指针表示字符串？

问题阐述

前面介绍过在定义字符串时其实就是定义了一个字符数组，那么如何用指针表示字符串呢？

专家解答

前面介绍的定义字符串的方式是下面的样式。

```
char charname[]="welcome to mr";
```

上面的语句就是定义了一个字符数组，数组的每一个元素保存一个字母，然后在最后一个元素中保存字符串的结束标志 `'\0'`，那么在操作这个字符串的时候，遇到字符串的结束标志后就会停止操作。

前面不止一次说到过，数组名就是数组的首地址，而上面也说到字符串实质就是一个字符数组，那么这样就不难想到使用指针如何表示字符串了。那么如何表示呢？来看一下。



定义一个字符串指针变量。

```
char *指针变量名
```

要注意的是，类型说明符必须是 `char`，那么上面的语句就可以写成下面的格式。

```
char *charname="welcome to mr";
```

上面的语句可以理解为定义一个指向字符串常量的指针，指针名是 `charname`。如果要对上面的常量进行输出，就可以使用下面的语句。

```
printf("%s",charname);
```

在 `printf()` 函数列表中，只要给出指针变量名就可以了。

专家点评

指针、数组和字符串之间有着密不可分的关系，读者应该充分理解它们的概念，这样操作它们就不难了。

问题 203 如何使用字符串指针作为函数参数？

问题阐述

前面讲过使用指针作为函数的参数，但都是数字型的，而且 C 语言中许多字符串操作通常是由指针运算来实现的，那么如何使用字符串指针作为函数参数呢？

专家解答

将字符串以实参的形式传递给函数的形参，结合上面问题介绍的内容，很容易就可以想到使用指针将字符串传递给函数。下面就来看一个示例程序，代码如下。

```
#include<stdio.h>
#include<stdlib.h>
int comp(char *s1,char *s2){
    return strcmp(s1,s2);
}
int main()
{
    int num;
    char str1[20],str2[20];
    printf("请输入一个 20 字节内的字符串: \n");
    scanf("%s",str1);
    printf("请再输入一个 20 字节内的字符串: \n");
    scanf("%s",str2);
    num=comp(str1,str2);
```



Note



Note

```

if(num==0){
    printf("您输入的字符串是相同的!\n");
}else if(num<0){
    printf("您输入的的第一个字符串小于第二个字符串\n");
}else {
    printf("您输入的的第一个字符串大于第二个字符串\n");
}
system("PAUSE");
return 0;
}

```

上面程序的运行结果如图 11.17 所示。

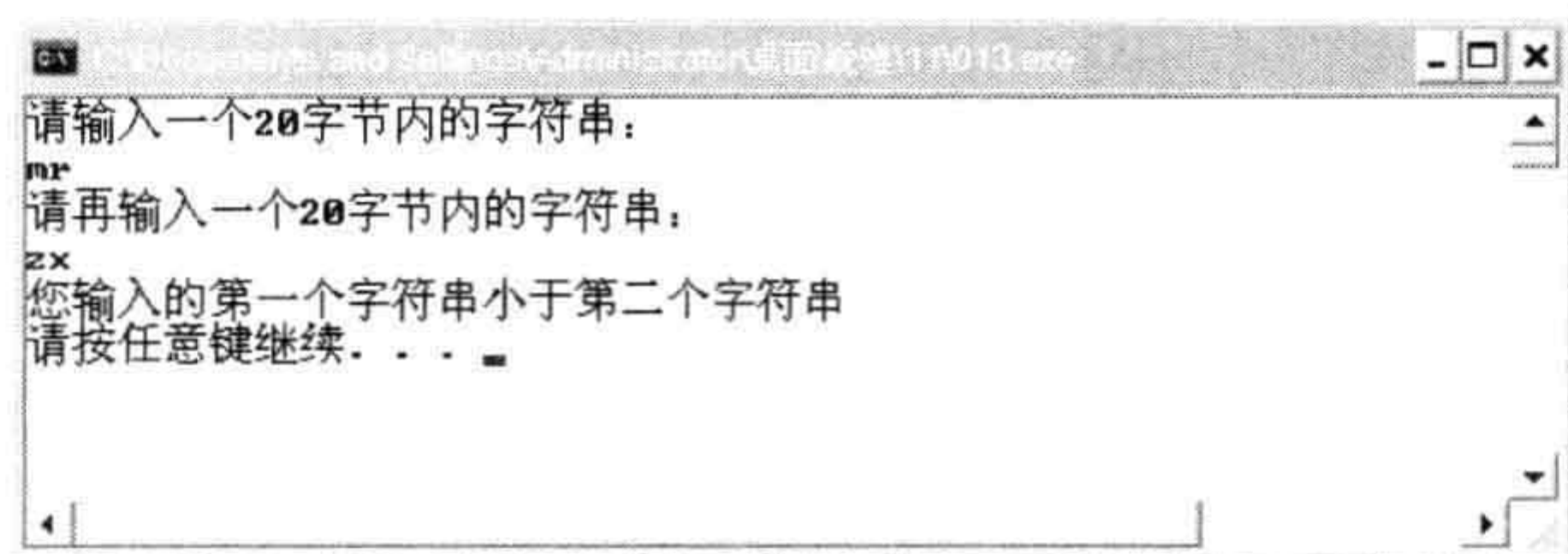


图 11.17 程序运行结果

上面程序中，在输入函数中并没有使用取地址符号&，而是直接使用数组名。在传递参数时，使用了指针变量，通过输出结果的验证，这样是可行的，而且正确。

专家点评

读者要注意在开发时不要拘泥于一种方式，而是要大胆地尝试和优化自己的程序。只有这样，才能写出高质量的程序代码。

问题 204 字符数组和字符指针的区别是什么？

问题阐述

前面交互地讲解了字符数组和字符指针的内容，那么它们有什么区别呢？

专家解答

从前面的一些实例可以看出，使用字符数组和指针变量都可以实现字符串的处理，两者之间看似没什么区别，但实际上它们之间是有很区别的。

从组成上来讲，字符数组是由若干个数组元素组成的，每个元素保存一个字符；而字符串指针变量本身是一个变量，它保存的是字符串常量的首地址，必须要初始化后才可以使用的。

前面曾多次说到，数组名是数组的首地址，是一个常量，一旦确定了存储位置后，就



不可以改变；而字符串变量本身就是一个变量，它保存的数据地址是可以改变的。

例如下面的语句。

```
char c[]="welcome to mr";  
char *s;  
s="welcome to mr";
```

其中，第一句就是定义一个数组，并初始化，s 就是数组名，是数组的首地址，但是如果进行 s++就是错的；而第二和第三条语句是一组，定义字符串指针变量 s，指向字符串“welcome to mr”，但是可以进行 s++运算，指向下一位。

专家点评

表面看是相同的，操作结果也一样，但不一定实质就是一样的，有时是有着天壤之别的，这一点读者要注意。

问题 205 什么是指针数组？

问题阐述

前面曾提到过数组指针，也提到过指针数组，数组指针介绍了，那么到底什么是指针数组呢？

专家解答

数组的元素值为指针就是指针数组。指针数组是指一组有序的指针的集合。指针数组的所有元素都必须是具有相同存储类型和指向相同数据类型的指针变量。

指针数组说明的一般形式为：

类型说明符 *数组名[数组长度]

其中，类型说明符为指针值所指向的变量的类型。

例如：

```
int *p[3]
```

上面语句表示 p 是一个指针数组，它有三个数组元素，每个元素值都是一个指针，指向整型变量。

专家点评

读者应注意尤其在排序时，采用普通的排序方法，逐个比较之后交换字符串的位置。交换字符串的物理位置是通过字符串复制函数完成的。反复地交换将使程序执行的速度很慢，又增加了存储管理的负担。用指针数组能很好地解决这些问题。把所有的字符串存放



Note



在一个数组中,把这些字符数组的首地址放在一个指针数组中,当需要交换两个字符串时,只须交换指针数组相应两元素的内容(地址)即可,而不必交换字符串本身。



Note

问题 206 如何使用指针数组处理字符串?

问题阐述

上面讲到了指针数组的概念,那么该如何使用指针数组处理字符串呢?

专家解答

使用指针数组可以很方便地处理多个字符串。前面讲到字符串实质可以使用一维数组来进行表示,也可以使用一个字符串指针指向一个字符串。例如:

```
char str[]="welcome to mr";  
char *str="welcome to mr";
```

首先来看一下使用数组进行处理字符串。先定义一个二维数组,然后每一个字符串作为一行,但是这样真的太浪费存储空间,因为对于二维数组来讲,每一行所占用的内存空间是相同的,这样就说明所定义的二维数组的行长度就是所要存储的多个字符串中最长字符串的长度,那么对于存储较短字符串的行,多余的空间就会空出,而造成浪费。

那么如果使用指针呢?先来看一个程序。

```
#include<stdio.h>  
#include<stdlib.h>  
int main()  
{  
    int i;  
    char *str[]={"c","java","c++","c#"};  
    for(i=0;i<sizeof(str)/4;i++){  
        printf("%s\n",str[i]);  
    }  
    system("PAUSE");  
    return 0;  
}
```

程序中存储字符串的指针数组只是将每个字符串的首地址进行了存储,而且所有字符串不需要存储在一片连续的内存中,所以每个字符串只需根据自己的实际长度存储就可以了。

专家点评

从上面的介绍可以得出,使用指针数组操作字符串可以节省大量的内存空间,对于程序的优化有着重要的作用。



问题 207 如何将指针数组作为函数的参数?

问题阐述

上面介绍了如何将字符串指针作为函数的参数,那么如何将指针数组作为函数的参数呢?

专家解答

使用字符指针作为函数的参数传递字符串,一般传递的只是一个字符串,如何传递多个字符串呢?有人会想到二维数组,但是上一问题中已经说到,二维数组存储多字符串太浪费存储空间,所以还是使用指针数组比较好,那么如何使用呢,下面给出一个实例。

```
char *day_name(char *name[],int n)
{
    char *p1,*p2;
    p1=*name;
    p2=*(name+n);
    return((n<1||n>7)? p1:p2);
}

main(){
    static char *day[]={"no day",
                        "Monday",
                        "Tuesday",
                        "Wednesday",
                        "Thursday",
                        "Friday",
                        "Saturday",
                        "Sunday"};

    char *p;
    int i;
    char *day_name(char *day[],int n);
    printf("输入星期数:\n");
    scanf("%d",&i);
    if(i<0) exit(1);
    ps=day_name(day,i);
    printf("星期数对应的英文写法:%2d-->%s\n",i,p);
}
```

实例程序主函数中定义了一个指针数组 `day`, 并对 `day` 作了初始化赋值。其每个元素都指向一个字符串。然后又以 `day` 作为实参调用指针型函数 `day_name`, 在调用时把数组名 `day` 赋予形参变量 `day`, 输入的整数 `i` 作为第二个实参赋予形参 `n`。在 `day_name` 函数中定义了两个指针变量 `p1` 和 `p2`, `p1` 被赋予 `day[0]` 的值(即 `*day`), `p2` 被赋予 `day[n]` 的值即 `*(day+`



Note



n)。由条件表达式决定返回 p1 或 p2 指针给主函数中的指针变量 p，最后输出 i 和 p 的值。

专家点评

节约存储空间是衡量一个程序的标准之一，尤其对于一些用户量大的程序，一个用户的少量浪费都将是整个系统最大的损失。

问题 208 什么是指向指针的指针？

问题阐述

一个指针变量可以指向整型变量、实型变量、字符类型变量，当然也可以指向指针类型变量，那么它到底是什么样子的呢？

专家解答

当指针变量用于指向指针类型变量时，则称之为指向指针的指针变量。这种双重指针如图 11.18 所示。

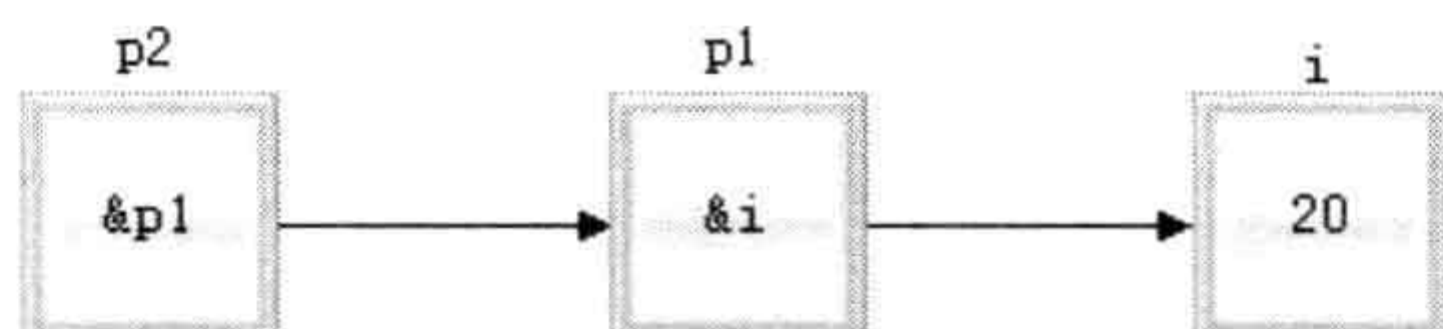


图 11.18 指向指针的指针

整型变量 i 地址是 &i，其值传递给指针变量 p1，则 p1 指向 x，同时，将 p1 的地址 &p1 传递给 p2，则 p2 指向 p1。这里的 p2 就是前面讲到的指向指针变量的指针变量，即指针的指针。指向指针的指针变量定义如下。

类型标识符 **指针变量名；

例如：

```
int **p;
```

其含义为定义一个指针变量 p，它指向另一个指针变量，该指针变量又指向一个基本整型变量。由于指针运算符 “*” 是自右至左结合，所以上述定义相当于：

```
int *(*p);
```

专家点评

对于指向指针的指针，使用第一个 * 运算符得到的是一个内存地址，使用第二个 * 取出数值，就是两个 *，可以得到所指变量的值。



问题 209 二级指针如何应用于一维数组?

问题阐述

上面介绍了指向指针的指针，即二级指针，那么如何使用二级指针操作一维数组呢？

专家解答

下面看一下指向指针变量的指针变量在程序中是如何应用的。
利用指向指针的指针输出一维数组。代码如下。

```
#include<stdio.h>
main()
{
    int a[10],*p1,**p2,i;           /*定义数组、指针、变量等为基本整型*/
    printf("please input:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);         /*给数组 a 中各元素赋值*/
    p1=a;                           /*将数组 a 的首地址赋给 p1*/
    p2=&p1;                          /*将指针 p1 的地址赋给 p2*/
    printf("the array is:");
    for(i=0;i<10;i++)
    {
        if(i%5==0)                  /*每输出 r 个元素进行一次换行*/
            printf("\n");
        printf("%5d\n",*(*p2+i));    /*输出数组中的元素*/
    }
}
```

程序运行结果如图 11.19 所示。

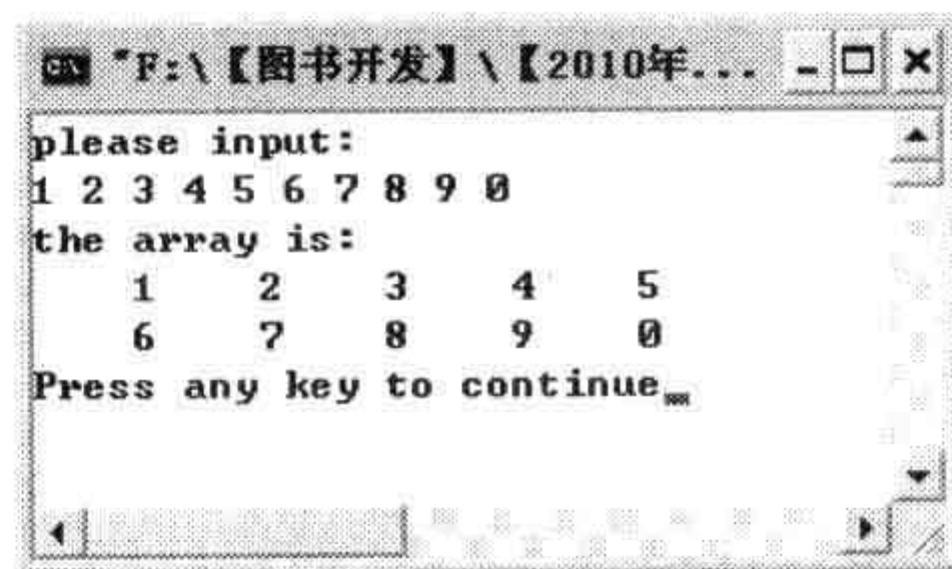


图 11.19 一维数组输出

该程序中，将数组 a 的首地址赋给指针变量 p1，又将指针变量 p1 的地址赋给 p2，要通过这个双重指针变量 p2 访问数组中的元素，就要一层层地来分析。首先看 *p2 的含义，*p2 指向的是指针变量 p1 所存放的内容及数组 a 的首地址，要想取出数组 a 中的元素，就必须在 *p2 前面再加一个指针运算符“*”。上面描述的过程如图 11.20 所示。





Note

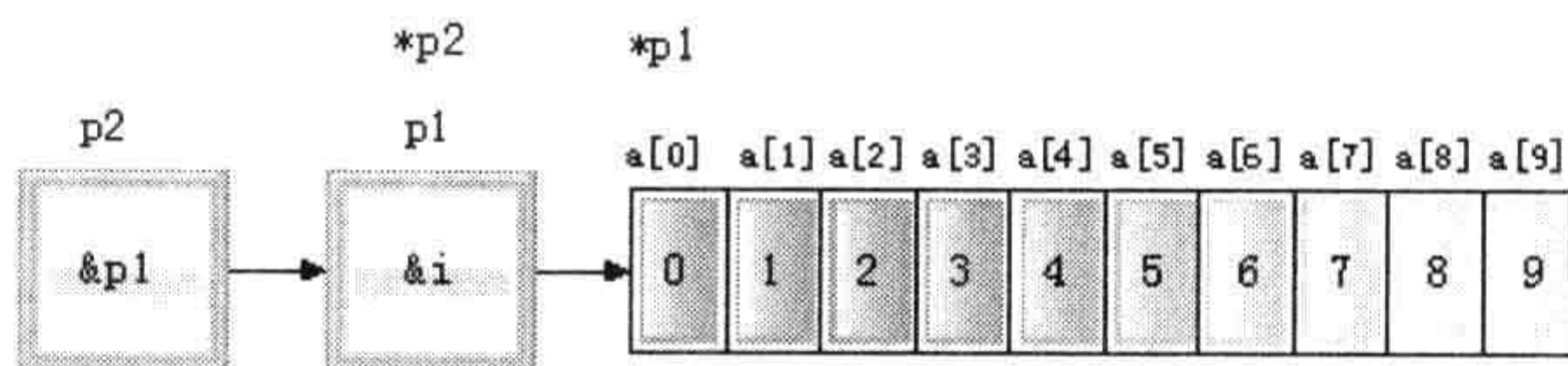


图 11.20 指向数组指针的指针

根据前面讲过的指针的用法，还可将程序改写成如下形式。

```
#include<stdio.h>
main()
{
    int a[10],*p1,**p2;                /*定义数组、指针等为基本整型*/
    printf("please input:\n");
    for(p1=a;p1-a<10;p1++)             /*指针 p 从 a 的首地址开始变化*/
    {
        p2=&p1;                        /*将指针 p1 的地址赋给 p2*/
        scanf("%d",*p2);               /*通过指针变量给数组元素赋初值*/
    }
    printf("the array is:");
    for(p1=a;p1-a<10;p1++)
    {
        if((p1-a)%5==0)                /*每输出 5 个元素实现一次换行*/
            printf("\n");
        p2=&p1;                        /*将 p1 地址赋给 p2*/
        printf("%5d",**p2);             /*将数组中的元素输出*/
    }
}
```

专家点评

指向指针的指针不仅仅是对象狭义上的指针，同样可以指向数组指针。

问题 210 如何实现二级指针对二维数组的操作？

问题阐述

上一问题讲解了如何使用二级指针对一维数组的操作，那么如何操作二维数组呢？

专家解答

要更清楚地了解二维数组的指针，首先要掌握二维数组数据结构的特性。二维数组可以看成是元素值为一维数组的数组。假设有一个 3 行 4 列的二维数组 a，它定义为：

```
int a[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

a 是数组名。a 数组包含 3 行，即 3 个元素：即 a[0]、a[1]、a[2]。而每个元素又是一



个包含 4 个元素的一维数组。同一维数组一样， a 的值为数组首元素地址值，而这里的首元素为 4 个元素组成的一维数组。因此，从二维数组角度看， a 代表的是首行的首地址。 $a+1$ 代表的是第一行的首地址。 $a[0]+0$ 可以表示为 $\&a[0][0]$ ，即首行首元素地址； $a[0]+1$ 可以表示为 $\&a[0][1]$ ，即首行第二个元素的地址。

使用指针指向数组时，在一维数组中 $a[0]$ 与 $*a[0]$ 等价， $a[1]$ 与 $*a(+1)$ 等价。因此，在二维数组中 $a[0]+1$ 和 $*(a+0)+1$ 的值都是 $\&a[0][1]$ ，如图 11.21 中地址 1002， $a[1]+2$ 和 $*(a+1)+2$ 的值都是 $\&a[1][2]$ ，如图 11.21 中的地址 1012。

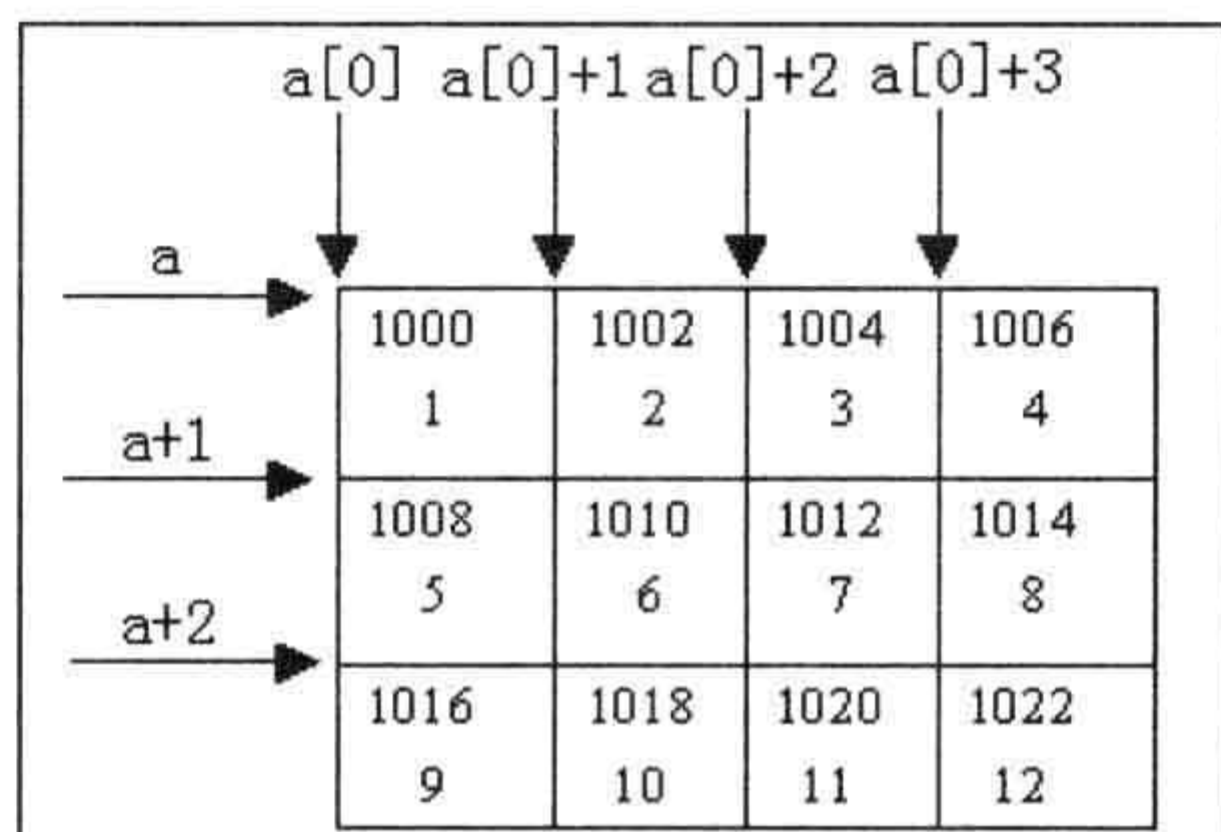


图 11.21 二维数组地址描述

下面再来看一个实例程序。

```
#include<stdio.h>
void main()
{
    int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};    /*声明数组*/
    printf("%d,%d\n",a,*a);                        /*输出第 0 行首地址和 0 行 0 类元素地址*/
    printf("%d,%d\n",a[0],*(a+0));                /*输出 0 行 0 列地址*/
    printf("%d,%d\n",&a[0],&a[0][0]);              /*0 行首地址和 0 行 0 列地址*/
    printf("%d,%d\n",a[1],a+1);                    /*输出 1 行 0 列地址和 1 行首地址*/
    printf("%d,%d\n",&a[1][0],*(a+1)+0);          /*输出 1 行 0 列地址*/
    printf("%d,%d\n",a[1][1],*(*(a+1)+1));        /*输出 1 行 1 列元素值*/
    getch();
}
```

专家点评

上面实例程序中给出了对二维数组的基本操作，那么读者可以根据这些基本操作的组合作出更高级的操作。

问题 211 二级指针如何操作字符串数组（指针数组）？

问题阐述

上一问题讲解了二级指针对基本数组的操作，那么对于字符串数组该如何操作呢（而且是使用指针数组存储）？





专家解答

使用指针的指针实现对字符串数组中字符串的输出。指向指针的指针即是指向指针数据的指针变量。这里创建一个指针数组 `strings`，它的每个数组元素相当于一个指针变量，都可以指向一个整型变量，其值为地址，如图 11.22 所示。

`strings` 是一个数组，它的每个元素都有相应的地址。数组名 `strings` 代表该指针数组的首单元的指针，就是说指针数组首单元中存放的也是一个指针。`strings+i` 是 `strings[i]` 的地址。`strings+i` 就是指向指针型数据的指针。

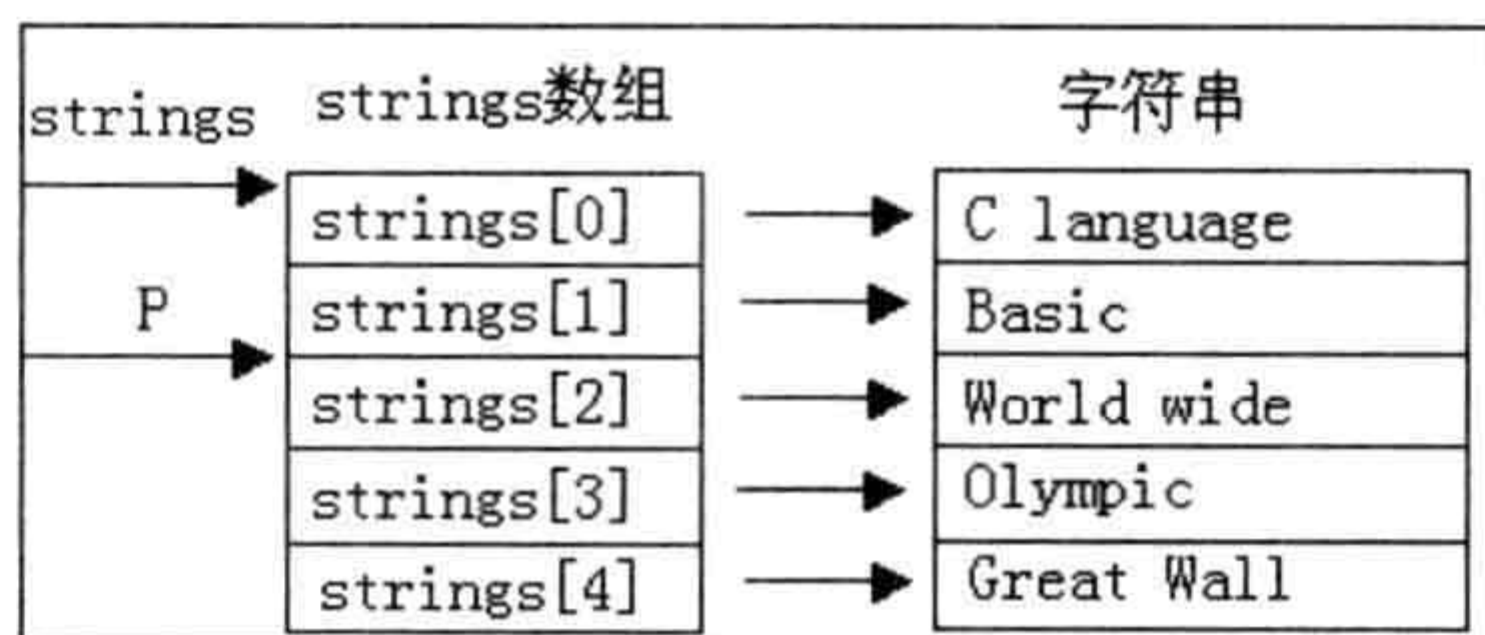


图 11.22 指针数组结构示意图

指向指针数据的指针变量定义语句形式如下。

```
char **P;
```

`p` 的前面有两个*号，*运算符是从右至左结合，**`p` 就相当于*(*`p`)，*`p` 表示定义一个指针变量，在其前面再添加一个*号，表示指针变量 `p` 是指向一个指针变量。*`p` 就表示 `p` 所指向的另一个指针变量，即一个地址。**`p` 是 `p` 间接指向的对象的值。例如，这里*(`p+2`)就表示 `strings[2]` 中的内容，它也是一个指针，指向字符串“World wide”。因此，输出字符串时，语句为：

```
printf("%s\n",*(p+i));
```

下面来看一个具体的程序，让大家对此有更进一步的了解。

```
main()
{
    char *strings[]={"C language",
                    "Basic",
                    "World wide",
                    "Olympic",
                    "Great Wall"};

    char **p,i;
    p=strings;
    for(i=0;i<5;i++)
    {
        printf("%s\n",*(p+i));
    }
}
```

/*使用指针数组创建字符串数组*/
/*声明变量*/
/*指针指向字符串数组首地址*/
/*循环输出字符串*/



专家点评

指针数组本身存储的就是字符串的首地址,再使用二级指针去进行操作,这很难理解,但是所带来的优越性能却是无可比拟的,所以大家一定要尽力去理解、掌握这一部分内容。



Note

问题 212 如何理解返回指针的函数?

问题阐述

有些函数的返回类型是指针类型,如何理解返回指针的函数?

专家解答

一个函数可以带回一个整型值、字符值、实型值等,也可以带回指针型的数据,即地址。其概念与以前类似,只是带回的值的类型是指针类型而已。返回指针值的函数简称为指针函数。

定义指针函数的一般形式为:

类型名 *函数名(参数表列);

例如:

```
int *fun(int x,int y)
```

fun 是函数名,调用它以后能得到一个指向整型数据的指针。x 和 y 是函数 fun 的形式参数,这两个参数也均为基本整型。这个函数的函数名前面有一个*,表示此函数是指针型函数,类型说明是 int 表示返回的指针指向整型变量。

指针函数实例,求长方形的周长。代码如下。

```
#include<stdio.h>
int per(int a,int b);
void main()
{
    int iWidth,iLength,iResult;
    printf("请输入长方形的长: \n");
    scanf("%d",&iLength);
    printf("请输入长方形的宽: \n");
    scanf("%d",&iWidth);
    iResult=per(iWidth,iLength);
    printf("长方形的周长是: ");
    printf("%d\n",iResult);
}
int per(int a,int b)
{
```




```
return (a+b)*2;
```

```
}
```

程序运行结果如图 11.23 所示。



Note

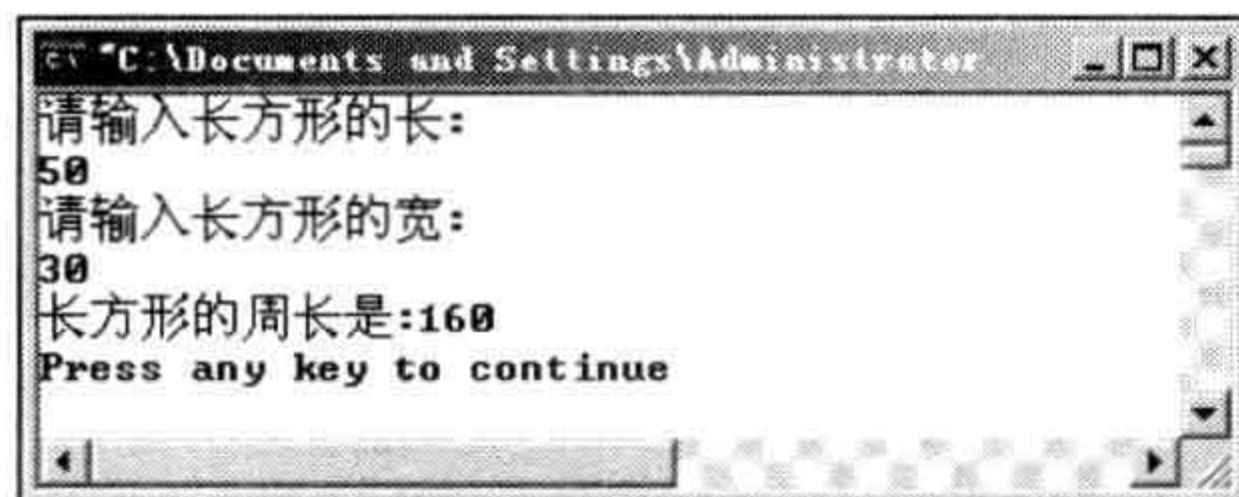


图 11.23 求长方形周长

上面程序用前面讲过的方式自定义了一个函数 `per()`，用来求长方形的面积，下面就来看一下在上面程序的基础上如何使用返回值为指针的函数。代码如下。

```
#include<stdio.h>
int *per(int a,int b);
int Perimeter;
void main()
{
    int iWidth,iLength;
    int *iResult;
    printf("请输入长方形的长: \n");
    scanf("%d",&iLength);
    printf("请输入长方形的宽: \n");
    scanf("%d",&iWidth);
    iResult=per(iWidth,iLength);
    printf("长方形的周长是: ");
    printf("%d\n",*iResult);
}
int *per(int a,int b)
{
    int *p;
    p=&Perimeter;
    Perimeter=(a+b)*2;
    return p;
}
```

程序中自定义了一个返回指针值的函数。

```
int * per(int x,int y)
```

专家点评

将指向存放着所求的长方形周长的变量的指针变量返回。注意这个程序本身并不需要写成这种形式，因为对这种问题像上面这样编写出的程序并不简便，这里这样写只是起到讲解的作用。



问题 213 什么是指向函数的指针?

问题阐述

指针可以指向普通数值、数组,还可以指向指针,那么可以指向函数吗?答案是可以,那么它是什么样的呢?

专家解答

一个函数在编译时被分配一个入口地址,这个地址就称为函数的指针。所以,可以使用指针变量指向一个函数,然后通过该指针变量调用这个函数。

指向函数的指针变量的一般形式为:

数据类型(*指针变量名)();

这里的数据类型是指函数返回值的类型。

例如:

```
int (*pmin)();
```

(*p)()表示定义一个指向函数的指针变量,它用来存放函数的入口地址。在程序设计过程中,将一个函数地址赋给它,它就指向那个函数。函数指针变量赋值可按如下方式进行书写:

```
p=min;
```

可见,在赋值时只给出函数名称即可,不必给出函数的参数。

在使用函数指针变量调用函数时,要写出函数的参数。例如:

```
m=(*p)(a,b);
```

下面来看一个应用的实例:比较大小。代码如下。

```
min(int a,int b)
{
    if(a<b) return a;           /*如果 a 小于 b 则返回 a*/
    else return b;             /*否则返回 b*/
}
void main()
{
    int(*pmin)();
    int a, b, m;
    pmin = min;
    printf("Please input two integer numbers: \n");
    scanf("%d%d", &a, &b);      /*输入两个值*/
    m = (*pmin)(a, b);         /*返回最小值*/
}
```



Note



```
printf("min=%d", m);  
getch();  
}
```

在 `mian()` 函数中实现定义指向函数的指针变量，并使次函数指针变量指向 `min()` 函数，将得到的结果输出在窗体上。

专家点评

对于函数指针要注意以下两点：

- (1) 不能对函数指针变量进行算术运算。
- (2) 在函数调用时，不要忘记指针变量名两边的括号。

问题 214 如何用 const 控制指针？

问题阐述

使用 `const` 关键字可以定义一个符号常量，那么如何使用 `const` 来控制指针呢？

专家解答

在定义指针变量的时候，也可以通过 `const` 关键字来限制对指针变量的值的修改，或者是限制对指针变量所指向数据的修改。例如：

```
const int *p
```

定义了一个指针变量，也可以写成如下形式。

```
int const *p
```

上面语句控制 `*p` 的值不能修改，`*p` 的值是指指针变量所指向的变量，也就是不可以使用 `*p` 来对它所指向的数据进行修改，但是可以使用赋值等方式直接通过指向变量的变量名来修改指向变量的数值。

由上可知 `const` 控制的是 `*p`，但是变量 `p` 没有被控制，所以也可以通过对 `p` 的修改，让它指向其他地址，实现对数据的修改。

再看这样的语句。

```
int * const p;
```

这里，关键字 `const` 控制的就是变量 `p` 了，所以变量 `p` 是不允许修改的，也就是指针 `p` 是一个指针常量，与数组名一样，而且在定义之初就要初始化，在以后的程序中将不能改变指针变量的指向。

专家点评

虽然语句 `int * const p` 中指针变量的值不允许修改，但其指向的变量的值是允许修改的。



问题 215 什么是“野指针”？

问题阐述

“野指针”是一个比较陌生的术语，那么它到底是什么呢？

专家解答

当程序里声明了一个指针而又没有给这个指针赋值，使其指向一个地址时，这样的指针就称为“野指针”。“野指针”会随意地指向一个地址。当对这个指针进行操作时，就有可能对野指针指向地址的数据进行更改。如果该块内存的数据是程序中的重要数据，数据被破坏，就会产生意想不到的后果，可能导致程序的崩溃。因此程序中禁止“野指针”的存在。

专家点评

“野指针”是在定义指针后没有对其进行初始化，或者指针指向的内存被释放，而指针没有被设置为 NULL。野指针随机地指向一个地址，使用这个指针进行操作时，就会更改该内存的数据，造成程序数据的破坏，严重威胁着程序的安全。。

问题 216 main()函数的指针数组形参是怎么回事？

问题阐述

在使用一些开发工具生成 C 语言文件时，主函数 `main()` 中会有参数，这个参数到底是怎么回事儿呢？

专家解答

`main()` 称为主函数，是所有程序运行的入口。`main()` 函数是由系统调用的，当处于操作命令状态下，输入 `main()` 所在的文件名，系统就调用 `main()` 函数。在前面课程的学习中，对 `main()` 函数始终作为主调函数处理，即允许 `main()` 调用其他函数并传递参数。事实上，`main()` 函数既可以是无参函数，也可以是有参函数。对于有参的形式来说，就需要向其传递参数。那么，`main()` 函数的形参的值从何处得到呢？由于其他任何函数均不能调用 `main()` 函数，不能调用自然也就无法向 `main()` 函数传递参数，只能由程序之外传递而来。这个具体的问题怎样解决呢？下面先看一下 `main()` 函数的带参的形式：

```
main(int argc, char *argv[])
```

从函数参数的形式上看，包含一个整型和一个指针数组。当一个 C 的源程序经过编译



Note



和链接后，会生成扩展名为.exe 的可执行文件，这是可以在操作系统下直接运行的文件。对于 main() 函数来说，其实际参数和命令是一起给出的，也就是在一个命令行中包括命令名和需要传给 main 函数的参数。命令行的一般形式为：

命令名 参数 1 参数 2...参数 n。

命令行中的命令就是可执行文件的文件名，其后所跟参数需用空格分隔，并作为对命令的进一步补充，也即是传递给 main() 函数的参数。命令行与 main() 函数的参数存在如下的关系。

设命令行为：

file1 str1 str2 str3

其中 file1 为文件名，也就是一个由 file1.c 经编译和链接后生成的可执行文件 file1.exe，其后各跟 3 个参数。以上命令行与 main() 函数中的形式参数关系如下。

它的参数 argc 记录了命令行中命令与参数的个数 (file1、str1、str2、str3)，共 4 个，指针数组的大小由参数的值决定，即为 char *argv[4]，该指针数组的取值情况如图 11.24 所示。

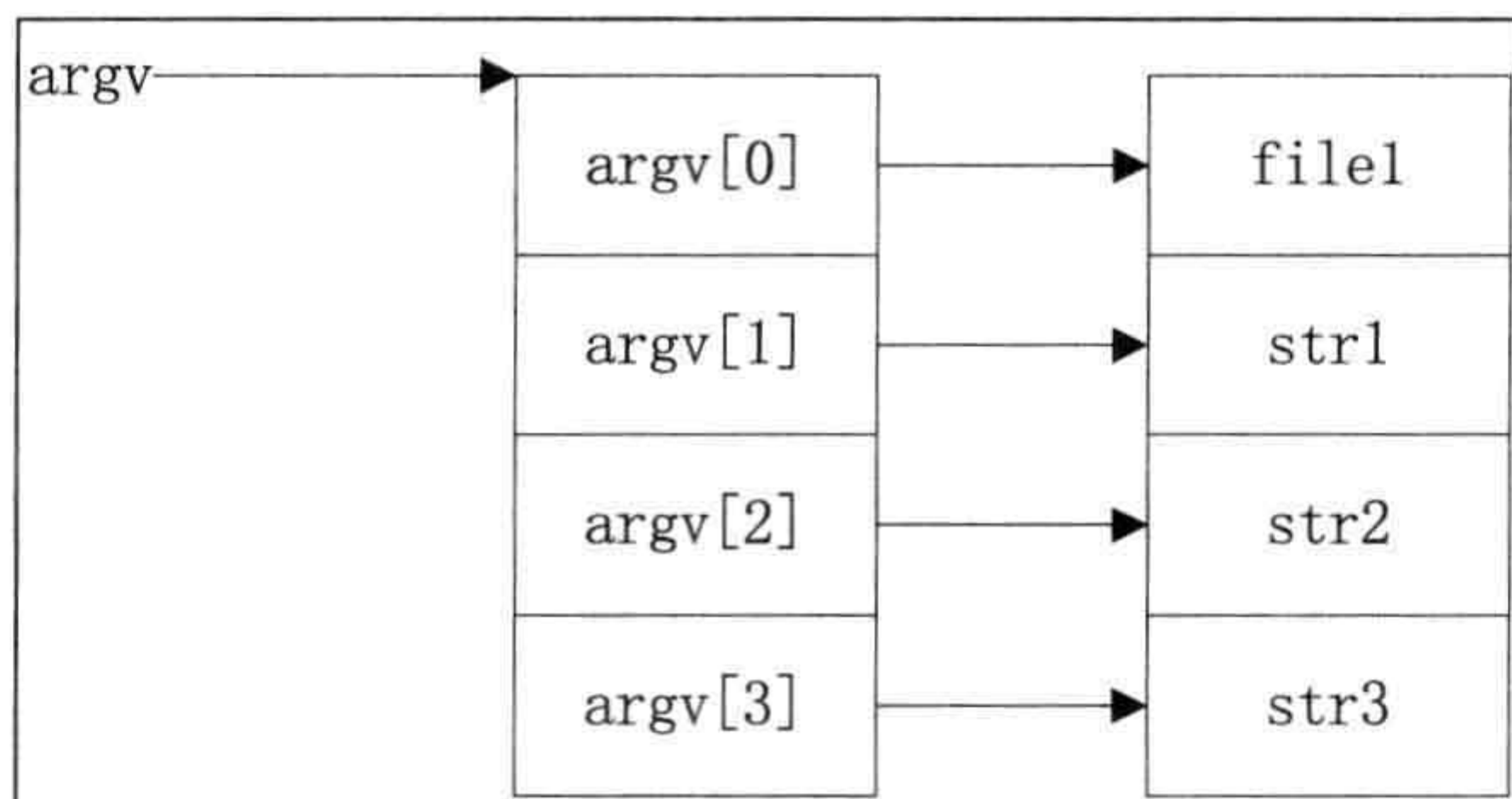


图 11.24 指针数组取值

输出 main() 函数参数内容。

```
#include<stdio.h>
main(int argc,char *argv[])                                /*main()函数为带参函数*/
{
    printf("the list of parameter:\n");
    while(argc>1)
    {
        ++argv;
        printf("%s\n",*argv);                                /*输出参数*/
        --argc;                                              /*argc 相应减 1*/
    }
}
```

输入内容如图 11.25 所示。

程序运行结果如图 11.26 所示。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 5.2.3790]
(C) 版权所有 1985-2003 Microsoft Corp.

C:\Documents and Settings\Administrator>d:\tc\0720 hello mingri I love China
```

图 11.25 输入命令行

```
C:\WINDOWS\system32\cmd.exe
Invalid keyboard code specified
the list of parameter:
hello
mingri
I
love
China
C:\DOCUME~1\ADMINI~1>
```

图 11.26 将输入内容输出



Note

专家点评

对于一般程序的开发，我们不会去关心 `mian()` 函数的参数，有时开发工具会自动生成，有时会手动删除，这些内容在程序编译时系统会自动进行相应的设置。

问题 217 void 指针就是空指针吗？它有什么作用？

问题阐述

这是一个在面试时很容易出现的问题，但是也是很多人混淆的问题，这个问题如何回答？

专家解答

`void` 指针一般称为通用指针，要与空指针严格区分。`void` 指针用于指向一个不属于任何类型的对象，所以 `void` 指针称为通用指针。

例如下面的声明。

```
int *p;
void *pt;
```

指针 `*p` 表示指向整型数据的指针；指针 `*pt` 表示所指向的对象不属于任何类型。

`void` 指针表示指向不属于任何类型的对象，它与空指针完全是两回事。

`void` 指针可以应用于函数指针，也可以应用于纯粹的内存操作。

专家点评

在指针和链表中使用非常广泛的空指针，表示的是不指向任何对象的一种指针。空指针通常定义一个常量 `NULL` 来表示，`NULL` 的值为 0。它是在头文件 `<stddef.h>` 中定义的一个宏，其值与任何有效的指针的值是不同的。`NULL` 是 0，它可能被强制转换为 `void *` 或者 `char *`，即 `NULL` 可能是 0 或者 `(void *) 0`。

空指针一般应用于以下三种情况：

- (1) 用空指针终止对递归数组结构的间接引用。
- (2) 用空指针作为函数调用失败时的返回值。
- (3) 用空指针作为警戒值。



注意:

不能间接引用一个空指针, 否则, 程序可能会得到一个毫无疑义的结果或者一个全是 0 的值, 而突然停止。



Note

问题 218 指针是一种特殊的变量, 只能用来保存地址。 这句话对吗?

问题阐述

对于这个问题, 答案是正确的, 那为什么呢?

专家解答

本题考查的是对指针概念的理解。指针类型不同于整型和其他的数据类型, 它是专门用来存放地址的数据类型。指针变量的定义形式为:

基类型 *变量名

例如:

```
int i;  
int *p;  
p=&i;
```

其中的 `p` 就是指针变量, 它的标志是变量名前有个 “*”, 表示该变量的类型为指针型变量。变量的名称是 `p`, 而不是 `*p`。`p` 指向变量 `i` 的地址, 所以在为 `p` 赋值的时候 `i` 要加上地址符, 此时 `p` 的前面不加 “*”。获取 `i` 的地址的时候直接使用 `p`, 获取变量 `i` 的值的时候要使用 `*p` 来获取。

专家点评

对于概念性的问题, 基本没有办法, 最好就是理解熟记。

问题 219 字符指针、浮点数指针以及函数指针这三种 类型的变量哪个占用的内存最大? 为什么?

问题阐述

字符指针、浮点数指针, 以及函数指针这三种类型的变量哪个占用的内存最大? 为什么?



专家解答

本问题就是指针与地址的关系的问题。指针就是用来保存地址值的，无论保存的是何种类型数据的地址，指针变量占用的内存都是一样的。

三者占用的内存一样大。因为指针变量也是一种变量，也占用内存单元，而且所有指针变量占用内存单元的数量都是相同的。不管是指向何种类型的指针变量，它们占用内存的字节数都是一样的，通常是一个机器字长。

专家点评

指针和地址的关系是一个贯穿整个指针部分的引绳式的问题，真正理解了它们之间的关系，也就理解了指针。

问题 220 一个 32 位的机器，该机器的指针是多少位？

问题阐述

一个 32 位的机器，该机器的指针是多少位？

专家解答

本问题考查的是对指针的理解，要清楚地了解指针，就要弄清楚指针与地址之间的关系。指针也属于一种数据类型，系统为指针变量分配一定的内存空间，用于存储指针指向的地址。请看下面的代码 `sizeof(p)` 的结果是多少。

```
char *p = malloc( 100 );           /*申请空间*/  
sizeof( p ) = ?
```

在 32 位机器中得到的结果是 4。因为无论指针变量指向何种类型的数据，指针变量的长度一般就是一个机器的字长。

指针变量的位数根据机器地址总线位数而定，对于 32 位地址总线的机器指针的位数就是 4 个字节。如果是 16 位系统那将不是这个答案，是多少？2 个字节。

专家点评

指针给编写程序提供了一个非常有用的工具，而且指针对有些程序来说是必不可少的，但是往往会由于一些疏忽而产生错误。在读操作中，错误的指针只是得到一些无用的存储内容。但在写操作中，可能将它错误地写到代码段或数据段。最关键的是这种错误只有在运行程序后才能表现出来。由于误用指针可能会造成严重的破坏，因此在使用时，一定要慎重地考虑，避免出现错误。

第12章

常用数据结构

- ▶▶ 空结构体所占的内存是多少?
- ▶▶ 在C语言中, 一个结构体可以包含指向自己的指针吗?
- ▶▶ `struct person{...}; person a;` 为什么编译出错?
- ▶▶ 怎样从/向数据文件读/写结构?
- ▶▶ 枚举与`#define`宏的区别有哪些?
- ▶▶ 如何看待枚举类型, 枚举类型的优点是什么?
- ▶▶ 关键字`typedef`的功能是什么?
- ▶▶ 类型定义是否允许嵌套?
- ▶▶ `typedef`与`#define`宏的相似之处与不同之处是什么?
- ▶▶ 什么是散列法?
- ▶▶ 大小端模式对`union`类型数据有什么影响?
- ▶▶ 如何为联合体变量赋初值?
- ▶▶ 如何证明联合体变量的所有成员是共享一个内存单元的?
- ▶▶ 堆和栈的区别是什么?
- ▶▶ 举例说明, 什么是静态链表? 什么是动态链表?
- ▶▶ 单向链表、双向链表和循环链表有什么区别?
- ▶▶ 如何在链表中的指定位置插入结点?
- ▶▶ 如何删除链表中指定位置的结点?
- ▶▶ 如何创建一个动态链表?
- ▶▶ 指向结构体数组的指针如何应用?



问题 221 空结构体所占的内存是多少?

问题阐述

标志结构体的关键字是 `struct`，它将一些相关联的数据封装成一个整体，方便在程序中使用。那么，空结构体所占内存是多少呢？

专家解答

1. 知识点介绍

结构体所占的内存大小是其成员所占内存之和，例如：

```
struct stu
{
    int num;
    char name[20];
    int score;
};
```

已知整型所占内存为 4 个字节，一个字符型所占内存为 1 个字节，因此这些结构体成员所占内存之和就是这个结构体所占的内存，为 28 个字节。

2. 上机验证

一个空结构体所占的内存有人会想到应该是 0，不占任何内存；有人会认为是 1，既然定义了，就至少有一个最小的内存，即 1 个字节。空想不如行动，亲自上机验证一下就知道答案了。

自定义一个 `employ` 空结构体，在 TC2.0 环境中运行一下，代码如下：

```
#include<stdio.h>
struct employ
{
}emp;
int main()
{
    printf("%d\n",sizeof(emp));
    return 0;
}
```

运行结果如图 12.1 所示。

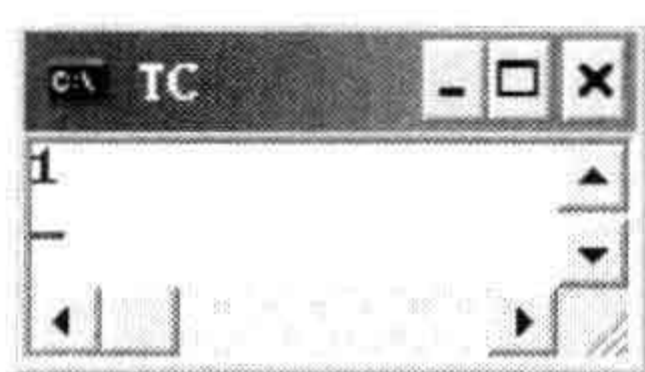


图 12.1 空结构体所占内存大小



Note



运行的结果竟然是 1，不是 0。因为编译器认为任何一种数据类型都有自己的大小，这样用这个类型定义一个变量才能分配确定的内存空间大小。结构体亦如此，无论里面的数据有多少，都占有自己确定的内存空间。那么，空结构体的内存空间不能为 0，应该为多大最理想呢？在所有的数据类型中 char 占内存最小，为 1 个字节，那么空结构体的内存应该在 0 到 1 范围内。然而，内存地址的最小单位是 1 个字节，不能为小数。那么，毋庸置疑，空结构体就只能定义为占 1 个字节。

专家点评

C 语言的知识体系比较庞大。在学习的过程中，有很多的细节问题不会被人注意到，例如刚才这个问题，很少有人遇到过，因此也很少有人会闲下来思考和验证这个问题。在学习 C 语言的过程中，希望程序员们不要忙于敲代码，要多思考，在思考的过程中，让现有的知识得到升华。

问题 222 在 C 语言中，一个结构体可以包含指向自己的指针吗？

问题阐述

```
typedef struct
{
    int num;
    short age;
    stu next;
}*stu;
```

上述这段代码为什么编译出错？一个结构体不可以包含指向自己的指针吗？

专家解答

在 C 语言中，一个结构体可以包含指向自己的指针，例如这样一个结构体类型：

```
struct person
{
    int id;
    char name[20];
    int age;
    struct person * next;
};
```

在建立链表操作时，用结构体变量做链表的结点，在结构体中需要至少有一个指针类型，用这个指针类型指向自己的结构体类型来存放下一个结点地址。由此可知，一个指针类型的成员既可以指向其他类型的结构体数据，也可以指向自己所在的结构体类型的数据。



既然一个结构体可以包含指向自己的指针，那么问题阐述中的代码错在哪里呢？

stu 例子的错误主要出在声明 next 的时候 typedef 还没有定义，可以先给这个自定义的结构体一个标签，如 struct student，然后声明 next 为 struct student *类型；或者对 typedef 类型和结构体类型分别定义。下面按照分别定义的方法对错误代码进行修改，代码如下。

```
struct student
{
    int num;
    short age;
    struct student *next;
};
typedef struct student *stu;
```

说明：

根据上述说明的修改方法，还可以写出几种不同的正确代码。

专家点评

在学习 C 语言时，有很多概念性的知识点要理解。typedef 定义是对一个数据类型定义别名，例如“typedef int COUNT;”就是将整型 int 定义为 COUNT，而数据类型 int 是已经存在的类型。那么当将 typedef 移植到自定义的结构体中定义时，需要首先定义结构体，然后再使用 typedef 对自定义的结构体进行别名定义。遇到编译错误，要寻找错误的答案，要分析问题，可以找同类问题的正确的代码进行比较，观察两者意义上的不同，而不是书写上的差异。

问题 223 struct person {...}; person a; 为什么编译出错？

问题阐述

在结构体中定义一个变量，可以有很多种方法，为什么这样定义编译出错呢？例如：

```
struct person {...}; person a;
```

专家解答

在解答编译出错的原因之前，先要了解几种正确的定义结构体类型变量的方法。

(1) 先声明，再定义。上面问题中的代码也是先声明，再定义，但是在定义变量的时候没加关键字 struct，因此编译出错。正确的定义结构体变量的形式如下。

```
struct person {...};
struct person a,b;
```



Note



`struct person` 为结构体类型名, `a` 和 `b` 为结构体变量名。在定义了结构体变量后, 系统会为之分配内存单元。

(2) 在声明类型的同时定义变量。与(1)的例子作用相同, 即定义了两个 `struct person` 类型的变量 `a` 和 `b`。例如:

```
struct person {...}a,b;
```

该定义方式的一般形式为:

```
struct 结构体名  
{  
    成员表列;  
}变量名表列;
```

(3) 直接定义结构体类型变量。直接定义结构体类型变量就是没有出现结构体名, 定义变量的一般形式为:

```
struct  
{  
    成员表列;  
}变量名表列;
```

掌握了上述正确定义变量的方法后, 来分析一下为什么题中所述的定义方法是错的呢?

`struct person {...}` 实质上是声明了一个“结构标签”, 即 `person` 只是这个结构体类型的一个标签, 真正的类型名为 `struct person`。因此, 在定义变量的时候, 需要完整的类型名来定义, 标签无法定义。

在了解了上述出错原因后, 再来考虑一下下面的声明与结构体的声明有什么不同? 例如:

```
typedef struct {...}A;
```

结构体类型的声明实质上是声明了一个“结构标签”, 而这个声明实质上是声明一个“类型定义”。声明了一个新的类型名 `A`, 此时的 `A` 不是变量, 而是一个此结构体类型的别名, 它代表上面指定的一个结构体类型, 这时可以用 `A` 定义一个变量, 例如:

```
A a,b;
```

注意:

上述代码不要写成 `struct A a,b;` 的形式。

专家点评

在学习结构体时, 要掌握并熟记定义结构体变量的几种方法, 并理解结构体声明的实质意义。了解 `typedef` 的意思就是声明一个新的类型名代替已有的类型名。



问题 224 怎样从/向数据文件读/写结构?

问题阐述

从数据文件读结构或者向数据文件写结构,都可以很轻松地使用文件读写函数实现,如使用 `fwrite()` 函数写一个结构,使用 `fread()` 函数读一个结构,但是这样读写出的文件却不能移植。怎么从/向数据文件读/写结构才能更好呢?

专家解答

使用 `fread()` 和 `fwrite()` 函数读写数据文件很常见,用来读写一个数据块,它们的一般形式为:

```
fread(buffer,size,count,fp);  
fwrite(buffer,size,count,fp);
```

`buffer` 是一个指针,对 `fread()` 函数来说是指向读入数据存放的地址,对 `fwrite()` 来说是指向要输出数据的地址;`size` 为要读写的字节数;`count` 为多少个 `size` 字节的数据项;`fp` 为文件指针。

说明:

如果文件以二进制形式打开,用以上这两个函数就可以读写任何类型的信息。

举例说明从数据文件读结构。例如,有一个自定义的结构体类型 `student` 为:

```
struct student  
{  
    char name[20];  
    int age;  
    int number;  
}stu[10];
```

结构体数组 `stu` 有 10 个元素,每一元素用来存放一个学生的数据(包括姓名、年龄、学号)。10 名学生的信息已经存储到指定的磁盘文件中,可以使用 `fread()` 函数读此结构信息。例如:

```
for(i=0;i<10;i++)  
    fread(&stu[i],sizeof(struct student),1,fp);
```

说明:

`fwrite()` 函数应用同样的方法可以向磁盘文件写入信息。

虽然应用上述函数可以向数据文件读写结构,但是这样写出的文件不能移植。最好的



Note



移植方法是用文本文件，使用 `fprintf()` 函数写入，使用 `fscanf()` 函数读入，或类似的函数。同时还应注意，如果结构包含任何指针，则只有指针值会被写入文件，当它们再次读回来的时候，很可能已经失效。最后为了广泛地移植，必须使用“b”标志打开文件。

移植性更好的方案是写一对函数，用可移植的方式按域读写结构，不过开始可能工作量稍大。

专家点评

在从数据文件读结构或者向数据文件写结构时，都应注意所使用的函数是应用于二进制文件还是文本文件，按照要求以相应的打开方式打开文件，否则容易出现乱码。

问题 225 枚举与#define 宏的区别有哪些？

问题阐述

枚举类型是 ANSI C 新标准中增加的，在 C 编译中，对枚举元素按常量处理，因此称枚举常量，它们不是变量，而 `#define` 是预处理中的宏定义，定义一个符号常量。那么，枚举与 `#define` 宏存在哪些区别呢？

专家解答

1. 枚举与#define 宏的概述

(1) 所谓枚举，是指将变量的值一一列举出来，变量的值只限于列举出来的值的范围内。例如，声明一个枚举类型 `weekday`：

```
enum weekday {sun,mon,tue,wed,thu,fri,sat};
```

C 语言编译按定义时的顺序使它们的值为 0,1,2...。可以输出枚举元素的值，也可以给枚举元素自定义值，后面的元素按顺序依次加 1。如定义 `sun=7`，`mon=1`，那么依次加 1 为 `tue=2`，`wed=3`，...，`sat=6`。

(2) `#define` 宏定义是用一个指定的标识符来代表一个字符串，例如：

```
#define PAI 3.14
```

此宏定义的作用是指定标识符 `PAI` 来代替“3.14”这个字符串，在编译预处理时，将程序中在该命令以后出现的所有的 `PAI` 都用“3.14”代替。它可以使用户能以一个简单的名字代替一个长的字符串。在程序中需要更改这个参数时，只需在宏定义中更改这个字符串就可以，从而降低出错率。

2. 枚举与#define 宏的区别

(1) 在编译器中可以调试枚举变量，但是不能调试宏常量。

(2) `#define` 宏常量是在预编译阶段进行简单替换。枚举常量则是在编译的时候确定其值。



(3) 枚举可以一次定义大量相关的常量，而#define 宏一次只能定义一个。

专家点评

其实学习一门编程语言很容易，但是在学习的过程中要学会随时整理已学的知识，将意思相近易混的知识做对比。只有对比之后，才会让原本零散的知识在脑中分类，知识体系才会更清晰，不会导致在学了很多的知识后，觉得自己知识学杂了，学乱了，越学越糊涂了。



Note

说明：

ANSI C 标准规定可以在 C 源程序中加入一些“预处理命令”(preprocessor directives)，以改进程序设计环境，这些预处理命令不是 C 语言本身的组成部分，不能直接对它们进行编译，必须在对程序进行通常的编译之前，先对程序中这些特殊的命令进行“预处理”，即根据预处理命令对程序做相应的处理。例如，#define 宏定义了一个符号常量 A，则在预处理时将程序中所有的 A 都置换为指定的字符串。

问题 226 如何看待枚举类型，枚举类型的优点是什么？

问题阐述

在众多数据类型中除了常见的整型、浮点型之外，还有结构体类型、共用体类型和枚举类型。对于枚举类型如何看待呢？枚举类型的优点是什么？

专家解答

1. 枚举类型的概述

所谓枚举类型，指的是凡属于该类型的变量的值都一一列举出来的一种数据类型。用枚举类型定义的变量的值只限于列举出来的值的范围内。声明一个枚举类型与声明一个结构体相似，其一般形式为：

```
enum 枚举名
{
    标识符[=整形常数],
    标识符[=整形常数],
    ...
};
```

如果枚举中的标识符没有初始化，则默认为从第一个标识符开始，顺次赋给标识符 0,1,2...。但当枚举中的某个标识符赋初值了，则其后边的标识符顺次加 1。

枚举类型声明结束后，要定义一个此枚举类型的变量，其一般形式为：

```
enum 枚举名 变量名;
```




枚举类型同结构体类型相似,也可以在声明枚举类型的同时定义变量。其一般形式为:

```
enum{...}变量名 1, 变量名 2;
```

2. 枚举类型定义变量的实例

定义一个颜色的枚举类型,例如:

```
enum color
{
    black=0;
    blue=1;
    red=2;
    yellow=3;
    green=4;
    orange=10;
    gray=11;
}c;
```

3. 枚举类型的优点

(1) 枚举常量相当于一个符号常量,因此具有见名知意的好处,可以增加程序的可读性。

(2) 枚举类型的变量取值范围限于列出的枚举常量范围,若取值不在列出的范围内,系统会视为出错。这样可以帮助系统检查错误,降低程序的理解难度。

(3) 枚举类型还便于系统对枚举变量进行类型检查,从而增强了安全性。

专家点评

枚举类型和结构体类型以及联合类型都是先定义数据类型,然后使用该数据类型定义变量。在某些情况下,枚举类型对程序还是很有帮助的。

问题 227 关键字 typedef 的功能是什么?

问题阐述

C 语言有 32 个关键字,其中 int 的功能是声明整型变量,struct 的功能是声明结构体变量,那么 typedef 的功能是什么呢?

专家解答

1. typedef 的功能

在 C 语言中除了可以使用标准类型名(如 int、char、float 等)和自定义结构体、共用体等类型外,还可以使用 typedef 给已有的类型定义别名,typedef 的功能就是给已经存在的数据类型取一个别名,注意并不是重新定义一个新的数据类型。



2. 为什么要给数据类型起别名

很多人都喜欢给别人“起外号”，typedef 关键字就好比是这个喜欢起外号的人。外号是随便起的吗？并不是，外号往往是根据这个人的特点而起的。例如，带眼镜的，外号就叫“小四眼”；如果长的高点，就叫“大个”；长的白点、胖点，可能就叫他“白猪”等。通过这些外号，人们可以更快地记住这些人（揭示别人缺点的外号可能会伤及他人的自尊，所以尽量不要给别人起外号！）。同样，给数据类型起外号也是有原因的，是为了更能突显出这个数据类型在这个程序中的作用。例如，int 型的数据在这个程序里主要起到了计数的作用，那么 typedef 就可以给这个 int 起个外号。例如：

```
typedef int COUNT;  
COUNT a,b;
```

当 COUNT 类型出现在程序中定义变量时，程序员们可以很清晰地判断出这是个用于计数的变量。这个外号使程序的可读性大大增强。

3. typedef 起别名引发的迷惑

在自定义结构体的数据类型中，会经常用到 typedef 为这个结构体取个别名，例如：

```
typedef struct PERSON  
{  
    int id;  
    int name[20];  
    char sex;  
    int age;  
}PER,*Per1;
```

为自定义的这个结构体取的别名为 PER 和 Per1，代表上面的这个结构体类型，定义一个此结构体类型的变量 a1，有如下方法：

(1) struct PERSON a1;

(2) PER a1;

这两种定义变量的方法是没有区别的。定义一个此结构体类型的指针的方法如下：

(1) struct PERSON *a2;

(2) Per1 a2;

(3) PER *s2;

上述三种定义结构体类型指针的方法也是没有区别的。

上述定义变量的方法很多初学者会迷惑，在此明确列出，希望可以减少在学习上的迷惑。其实，在定义结构体类型的变量 a1 时，可以将结构体 struct PERSON {...} 看成一个整体，typedef 就相当于给 struct PERSON {...} 起了个别名叫 PER；与此同时，给 struct PERSON {...} * 起了个别名叫 Per1。

专家点评

很多人在学习 typedef 时都会将其误认为是定义新的数据类型。其实不然，只是起个



Note



别名而已。



Note

问题 228 类型定义是否允许嵌套?

问题阐述

在 if 语句中又包含一个或多个 if 语句,称为 if 语句的嵌套。一个循环体中又包括另一个完整的循环结构,称为循环的嵌套。同样,函数也存在嵌套调用。了解了这么多嵌套之后,类型定义是否允许嵌套呢?

专家解答

1. 知识点概述

typedef 关键字代表类型定义,就是给一个已经存在的数据类型重新定义一个新名字,数据类型包括整型、字符型、结构体、联合体和枚举类型等。

类型说明的一般形式为:

```
typedef 类型 定义名;
```

用类型定义说明一个结构体的一般形式有:

```
typedef struct
{
    数据类型 成员名;
    数据类型 成员名;
    ...
}结构体名;
```

2. 疑难解答

用 typedef 类型定义为数据类型取的别名是允许嵌套的,意思就是用 typedef 类型定义为一个数据类型取了个别名,可以再应用 typedef 为这个别名再取个新别名,这就称之为类型定义的嵌套。例如:

```
typedef struct
{
    int grade;
    int class;
    int number;
    char name[20];
    char sex;
    float score;
}STUDENT;
STUDENT aa;
typedef STUDENT *P;
P p1,p2;
```




STUDENT 就是 typedef 为这个结构体取的别名, 用这个别名定义了一个变量 aa。进而又使用 typedef 为 STUDENT*取了个别名 P, 然后用这个结构体类型的指针定义了两个指针 p1 和 p2。这里的 p1 和 p2 是指向结构体类型变量的指针, 因为 STUDENT 被定义为这个结构体类型, 而 P 又被定义为 STUDENT 型的指针, 所以 P 定义的变量是指向结构体类型的指针。

专家点评

嵌套的目的是为了让程序更加清晰, 增加程序的可读性, 切勿盲目嵌套, 增加程序理解的难度, 以显示编程的能力。



Note

问题 229 typedef 与#define 宏的相似之处与不同之处是什么?

问题阐述

typedef 关键字的功能就是给一个已经存在的数据类型起一个在本程序中能够体现实际作用的名字。#define 宏定义是为一个字符串起一个别名, 在程序中应用到该字符串时, 用这个别名来替代。那么 typedef 与#define 的相似之处是什么, 不同之处又是什么呢?

专家解答

1. #define 与 typedef 的相似之处

宏定义和类型定义有很多相似之处, 通常都可以理解为为一个字符起别名, 在程序中用一个新的字符代替原有的字符。例如:

```
#define STRINT char
typedef char STRING;
```

这两句代码都是用 STRING 代替原有的字符 char。而 STRING str;的含义是定义了一个 char 型的变量 str。

在此例中, 宏定义和类型定义起到的作用是相同的, 但实质上是有区别的。

2. 两者的区别

(1) 在上述为 char 起别名的两种方法中, 实质含义是不同的。例如, #define 是在预编译时用 STRING 这个字符串代替 char 这个字符串;而 typedef 是为 char 字符串类型定义了一个别名, 并不是简单的字符串替换。STRING 这个字符串与 char 的功能相同, 代表了一个字符串类型的数据类型。

请看下面的例子, 观察两者的区别。

```
#define INTP int*
typedef int* INTP;
```




其中“INTP a,b;”在宏定义的定义下，可以理解成“int *a,b;”，表示声明了一个整型指针变量 a 和一个整型变量 b；若在类型定义的定义下，可以将其理解成“int *a;int *b;”，表示声明了两个整型变量指针 a 和 b。

(2) 宏定义和类型定义还有一个很大的区别，然而却往往被人忽略。那就是宏定义的句尾没有分号作为结束语句的标志；若有，就会被当做字符串替换进程序中。由此可以更明显地看出，宏定义通常被用作定义常量，以及用来实现“表面似和善，背后一长串”的宏，而且其本身并不在编译过程中进行，而是在预处理过程中就已经完成了。因此，很难发现潜在的错误及代码维护问题。例如，代码“#define PI 3.1415926;”只在程序源代码中应用到 PI 时才会出现编译错误，错误就在于用 PI 这个字符代替了“3.1415926;”这个字符串，比正常的数字多了个分号。

专家点评

#define 原本在 C 语言中是为了定义常量，然而随着 C++ 中的 const、enum 和 inline 的出现，使它渐渐地变成了起别名的工具。但是，在定义别名这个功能上，最好还是习惯应用 typedef。为了尽可能地兼容，一般都遵循#define 定义“可读”的常量（常量表达式），而 typedef 则常用来定义关键字或冗长的数据类型 的别名。

问题 230 什么是散列法？

问题阐述

什么是散列法？

专家解答

散列法是一种将字符组成字符串，转换为固定长度（一般是更短长度）的数值或索引值的方法，也叫哈希法，又可以称为杂凑法或关键码—地址转换法。

那么，通过散列函数得到的散列地址与原始关键码是一一对应的吗？

答案是否定的，即不是一一对应的，如图 12.2 所示。

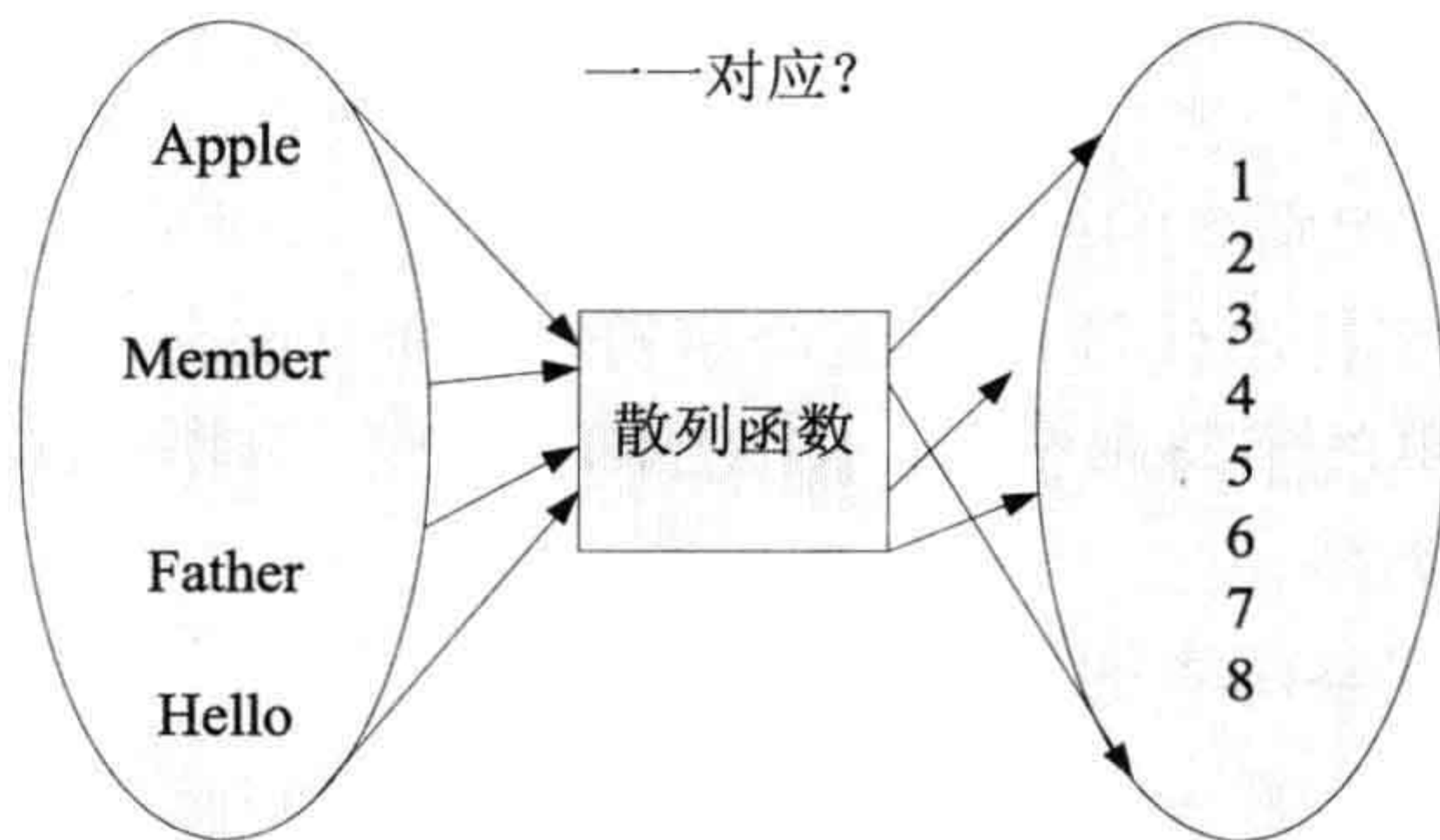


图 12.2 关键码与散列地址不是一一对应



一般来说,散列函数是一个压缩映像函数。通常关键码集合比散列表地址集合大得多。因此,有可能经过散列函数的计算,把不同的关键码映射到同一个散列地址上,这就不是——对应的了。于是就产生了冲突。如果表项按计算出的地址加入散列表时产生了冲突,必须再找一个地方来存放它,这就产生了解决冲突的问题。因此就要构造一个不容易产生冲突的函数,即构造一个地址分布比较均匀的散列函数,使关键码集合中的任何一个关键码经过这个散列函数的计算,映射到地址集合中所有地址的概率相等,以减少冲突。但实际上,由于关键码集合比地址集合大得多,冲突很难避免,所以对于散列方法需要注意两个问题:

- (1) 对于给定的一个关键码集合,选择一个计算简单且地址分布比较均匀的散列函数,避免或者尽量减少冲突;
- (2) 拟定解决冲突的方案。

专家点评

散列法一般用于在数据库中建立索引并进行搜索,同时还用于各种解密算法中。

问题 231 大小端模式对 union 类型数据有什么影响?

问题阐述

计算机都是以八位一个字节为存储单位的,所以一个 16 位的整型就存在两种可能的存储顺序:大端模式和小端模式。那么大小端模式对共用体类型中的数据存储又有什么影响呢?

专家解答

1. 大小端模式概述

考虑一个 int 型整数 29,将此整数转换为十六进制数为 0x3239,其中 0x32 为高位,0x39 为低位。那么这个十六进制数在内存中可能有两种存储方式:

(1)大端模式存储。高位 0x32 存放在低地址 0x1000 中,低位 0x39 存放在高地址 0x1001 中。结果如图 12.3 所示。

(2)小端模式存储。高位 0x32 存放在高地址 0x1001 中,低位 0x39 存放在低地址 0x1000 中。结果如图 12.4 所示。

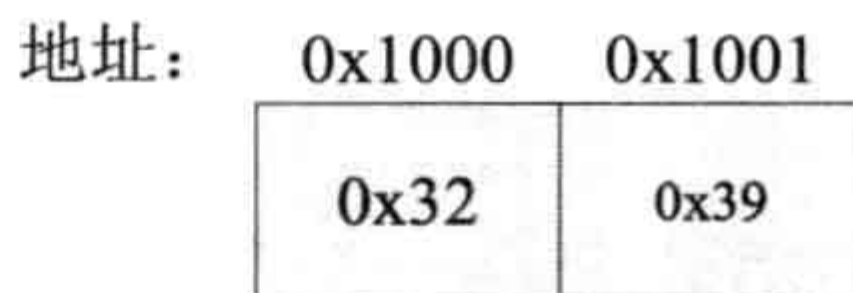


图 12.3 大端模式

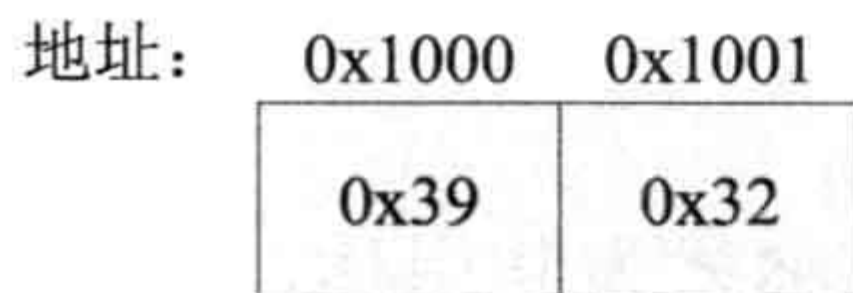


图 12.4 小端模式



大端模式 (Big_endian) 的含义就是字数据的高字节存储在低地址中, 而字数据的低字节则存放在高地址中。小端模式 (Little_endian) 的含义就是字数据的高字节存储在高地址中, 而字数据的低字节则存放在低地址中。

2. 举例说明大小端模式对共用体类型数据的影响

union 型数据所占的内存等于其最大的成员所占的内存, 对 union 型的成员的存取都是相对于该联合体基地址的偏移量为 0 处开始, 也就是联合体的访问无论对哪个变量的存取都是从 union 的首地址位置开始, 因此大小端模式的存储将会直接影响 union 成员的值。例如, 在主函数中声明一个联合体类型 u, 定义一个 union u 类型的变量 t, 引用 t 中的整型变量 i, 为其赋值为 1, 若引用字符输出, 显示 1 说明此系统的存储方式是小端模式, 输出 0 说明该系统的存储方式是大端模式。相应代码如下。

```
main()
{
    union u
    {
        int i;
        char ch;
    }t;
    t.i=1;
    printf("%d\n",t.ch);
}
```

程序运行结果如图 12.5 所示。

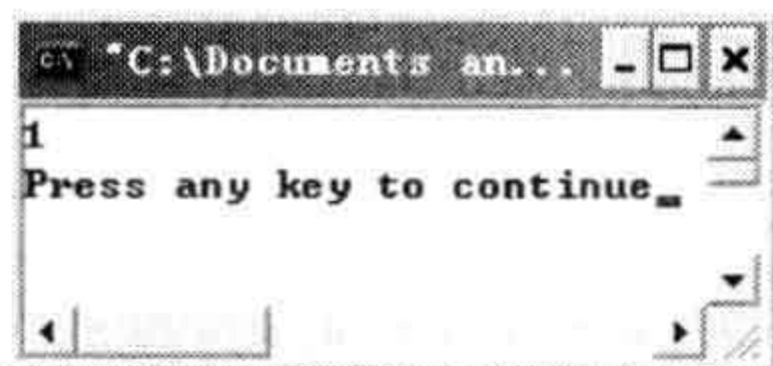


图 12.5 大小端模式对 union 数据的影响

专家点评

上述例子还可以验证当前系统的存储模式。由运行结果可以看出, 当前系统的存储模式为小端模式, 即高字节存放在高地址中, 低字节存放在低地址中。

问题 232 如何为联合体变量赋初值?

问题阐述

联合体又称之为共用体, 声明一个共用体类型, 必然要定义一个共用体类型的变量, 并对其赋初值。那么, 如何为共用体变量赋初值呢?



专家解答

1. 联合体类型定义变量

定义联合体类型变量的方法与定义结构体类型变量的方法相同，定义联合体类型变量的一般形式为：

```
union 联合体名
{
    成员表列
}变量表列;
```

举例说明定义联合体类型变量的形式，例如：

```
union number
{
    int i;
    char c;
    float f;
}m,n;
```

除了上述定义方法，还有两种定义变量的方法，与结构体类型定义变量的方法相同。

2. 联合体变量赋初值

只有先定义了联合体变量，才能引用联合体类型中的成员，不能引用联合体变量。例如，上述代码中定义了 *m* 和 *n* 为联合体变量，则存在如下引用成员的方式：

```
m.i;          /*引用联合体变量中的整型变量 i*/
m.c;          /*引用联合体变量中的字符型变量 c*/
m.f;          /*引用联合体变量中的浮点型变量 f*/
```

由于联合体变量具有所有成员共享一个内存地址的特点，因此为联合体变量赋初值时只能给该变量的第一个成员赋初值，其他成员不能赋初值。例如：

```
union number
{
    int i;
    char c;
    float f;
}m={2};
```

为联合体变量 *m* 中的整型变量 *i* 赋初值为 2。

专家点评

联合体的声明形式与结构体类型相似，但它们实质上是存在很大不同的。结构体类型所占内存空间是各个成员变量所占空间之和，而联合体类型所占的内存空间是成员变量中占内存最大的一个变量的内存，所有成员变量共用一个内存。



Note



问题 233 如何证明联合体变量的所有成员是共享一个内存单元的?

问题阐述

联合体又名共用体,用共用体这个名字更能体现这个类型的特点,那就是所有成员变量共用一个内存单元。联合体变量的这一特点决定了联合体变量的使用方法,即在某一时刻只有一个成员有意义,其他成员无意义。那么,怎么证明联合体的这一特性呢?

专家解答

为了证明联合体类型的成员变量是共享一个内存的,下面通过一个程序来进行说明。例如,声明一个联合体类型有两个成员变量,一个是 `int` 型的数组,另一个是 `char` 型的数组。先按照其中一个 `int` 型数组成员赋值,然后再按另一个 `char` 型数组成员输出。若整型数组与字符型数组是共享一个内存的,那么输出的内容与写入的内容应该是一致的,否则结果不一致。相应程序代码如下。

```
main()
{
    union ss
    {
        int a[6];
        char str[12];
    }u;
    u.a[0]=0x6548;      /*eH*/
    u.a[1]=0x6c6c;      /*ll*/
    u.a[2]=0x216f;      /*!o*/
    u.a[3]=0x0000;      /*NULL*/
    printf("%s\n",u.str);
}
```

运行结果如图 12.6 所示。

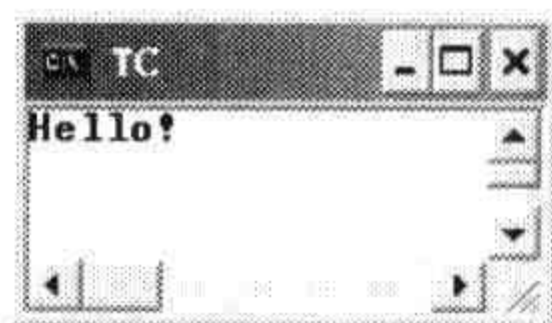


图 12.6 联合体中数组成员值

由此程序可以看出,每个整型变量占两个字节,因此整型数组的一个位置中存放两个字符。分别将两个字符的 ASCII 码值转换成二进制数赋给数组成员,然后以字符串的形式输出。结果与整型数组中存放的二进制数所表示的字符相同,所以联合体成员的变量共享一个内存空间这个特性得到证实。



专家点评

在给整形数组赋值时, 需要考虑联合体类型的大小端模式。联合体类型的存放顺序是从低字节到高字节。将高字节存放在高地址中, 将低字节存放在低地址中, 即所谓的小端模式。



Note

问题 234 堆和栈的区别是什么?

问题阐述

栈是存放函数的所有动态局部变量及函数调用和返回有关信息的一块内存。它为调试程序提供了一种功能, 能够显示一个已被调用的函数的列表, 这是调试程序时一种很有用的手段。

堆提供 `malloc()`, `calloc()` 和 `realloc()` 等函数获取内存空间的一块内存。下面采用对比的形式分别介绍堆和栈的作用, 以及它们的区别。

专家解答

1. 堆和栈的作用

(1) 栈的内存管理严格遵循先进后出的顺序, 即释放栈中对象所占内存时的顺序刚好与为这些对象分配栈中内存时的顺序相反, 这一点正是实现函数调用所需要的。从栈中分配内存效率特别高, C 语言编译程序能产生如此好的代码的原因之一就是充分利用了栈。

(2) 从堆中获取内存比从栈中获取内存要慢得多, 但是堆的内存管理却比栈灵活得多。任何时候都可以从堆中获取内存, 而且在释放堆中对象所占内存时, 可以按任意顺序进行。数据对象使用栈中内存比使用堆中内存程序运行更快。但是, 有时候使用堆中内存可以改善一种算法, 从而使它更快, 或者更灵活, 因此使用堆或栈要折中考虑。

2. 堆和栈的区别

在了解了堆和栈各自的用途后, 从两者的申请方式、申请后系统的响应、申请大小的限制、申请效率的比较和存储内容各方面了解它们的实质区别。

(1) 申请方式: 栈是由系统自动分配。如声明在函数中的一个局部变量 “`int i;`”, 系统会自动在栈中为 `i` 开辟内存空间。而堆是程序员自己申请, 同时指明大小。在 C 语言中, 堆提供 `malloc()`, `calloc()` 和 `realloc()` 等函数获取内存空间的一块内存, 如 “`buffer=(int*) malloc(256)`” 是为 `buffer` 分配了一个 256 字节大小的 `int` 型内存空间。

(2) 申请后系统的响应: 只要栈的剩余空间大于所申请空间, 系统将为程序提供内存, 否则将报异常, 提示栈溢出。而堆则是首先确定操作系统有一个记录空闲内存地址的链表。当系统收到程序的申请时, 会遍历此链表, 寻找第一个大于所申请空间的堆结点, 然后将该结点从空闲结点链表中删除, 并将该结点的控件分配给程序。

(3) 申请大小的限制: 在 windows 下, 栈是向低地址扩展的数据结构, 是一块连续的内存区域。它的最大容量是预先规定好的, 如果申请的空间超过栈剩余的空间, 将会提



示溢出，能从栈获得的空间较小。而堆是向高地址扩展的数据结构，是不连续的内存区域。这是因为系统是用链表来存储空闲内存地址的，而链表的遍历方向是由低地址向高地址，堆的大小受限于计算机系统中有效的虚拟内存。因此，堆获得的空间比较灵活，也比较大。

(4) 申请效率：在申请方式中，已经得知栈是由系统自动分配的，速度比较快，不容易控制。而堆是由 new 分配的内存，速度比较慢，用起来较方便。

(5) 存储内容：在函数调用时，第一个进栈的是主函数中的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数。在大多数 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意，静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。而堆一般是在堆的头部用一个字节存放堆的大小，堆中的具体内容程序员安排。

专家点评

通过对栈和堆的比较，大致了解了堆和栈的区别，以及用途。其实，栈和堆的关系可以理解为机洗衣服和手洗衣服。全自动洗衣机洗衣服，将所有的脏衣服放进去，倒上水，放上洗衣液就可以了。这样很便捷，但是灵活度和自由度较小。手洗衣服虽然很累，但是可以将自己认为比较脏的局部地方多洗洗或分类洗，灵活度较大。

问题 235 举例说明，什么是静态链表？什么是动态链表？

问题阐述

链表是用一组任意的存储单元存放线性表的元素，用指针表示元素间的逻辑关系。它是一种重要的数据结构，通常用到的都是动态链表。有动就有静，那什么是静态链表？什么又是动态链表呢？

专家解答

首先建立一个简单的链表，由三个学生信息的结点组成，输出各结点中的数据。相应代码如下。

```
#include<stdio.h>
#define NULL 0
struct student                                /*学生结构体*/
{
    int number;                                /*数据域*/
    struct student *link;                      /*指针域*/
};
main()
{
```




Note

```

struct student stu1,stu2,stu3,*head,*p;
stu1.number=101;           /*初始化结构体数据*/
stu2.number=102;
stu3.number=103;
head=&stu1;                 /*指定头指针*/
stu1.link=&stu2;            /*定义后继结点*/
stu2.link=&stu3;
stu3.link=NULL;            /*定义尾结点*/
p=head;                    /*使指针指向头结点*/
do
{
    printf("%d\n",p->number); /*输出数据域信息*/
    p=p->link;                /*指针指向下一个*/
}while(p!=NULL);
}

```

在此例中,学生结构体是结点,每一个结点都是在程序中定义的,空间不是临时开辟的,并且用完这些已经定义的空间后也不用释放,这就是所谓的静态链表。

程序的运行结果如图 12.7 所示。

相对于静态链表,那么动态链表又是什么呢?

动态链表,顾名思义,就是动态地分配内存的链表,即在需要时才开辟一个结点的存储单元。应用静态链表能够存储已知结点数的信息,当不确定会有多少个结点信息时,就会采用动态链表,这样节省内存。在动态链表中分配了内存,不用时必须释放内存,那么怎样分配内存和释放内存呢?在 C 语言中提供了以下相关函数。

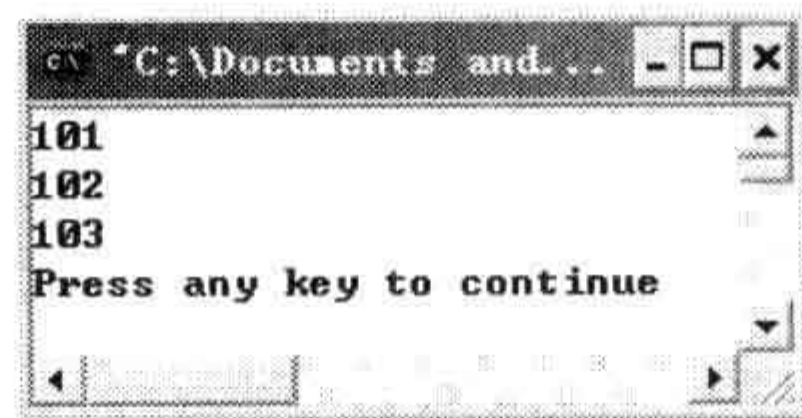


图 12.7 静态链表运行结果

(1) malloc()函数: size 为函数的参数。函数作用是在内存的动态存储区中,分配一个长度为 size 的连续空间。此函数的返回值是一个指向分配域起始地址的指针,如果此函数执行失败,返回空指针 (NULL)。

(2) calloc()函数: n 和 size 为函数的参数。函数功能是在内存的动态存储区中分配 n 个长度为 size 的连续空间,函数返回值同 malloc()函数。calloc()函数可以为二维数组开辟内存, n 为数组元素,每个元素长度为 size。

(3) free()函数: 指针 p 是函数的参数,函数用于释放由 p 指向的内存区,使这部分内存区能被其他变量使用。

专家点评

在链表应用中,通常用到的都是动态链表,能够动态地开辟内存单元,在使用结束后还可以释放此内存,方便其他变量使用,这样可以节省内存资源。



问题 236 单向链表、双向链表和循环链表有什么区别?

问题阐述

链表分为单向链表、双向链表和循环链表，它们的不同之处是什么呢?

专家解答

(1) 单向链表。所谓单向链表，就是指数据结点是单向排列的。一个单向链表结点由两个域组成，存储在结构体类型中。一个域用于存放数据元素 `data`，其数据类型由应用问题决定，称为数据域；另一个域存放一个指向该链表中下一个结点的指针 `link`，指向下一个结点的开始存储地址，称为链域或者指针域。例如，可以写成这样一个结构体类型。

```
struct NODE
{
    int number;
    char name[20];
    struct student *next;
};
```

其中 `number` 和 `name[20]` 用来存放数据域中的数据信息；指针 `*next` 用来存储下一个结点的指针，就是指针域部分。

图 12.8 为一个简单的单向链表。

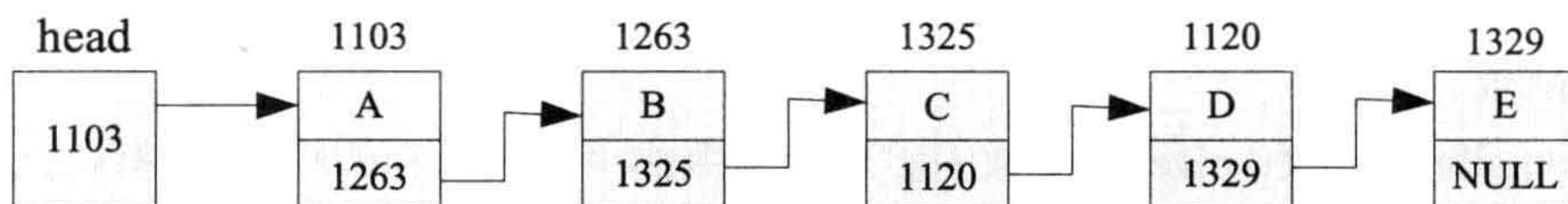


图 12.8 单向链表

单向链表有一个头结点，在图 12.8 中以 `head` 表示，它存放头结点的地址，没有数据，只是一个指针变量，又叫做“头指针”。链表中的其余结点都有两个部分，即数据域和指针域。在 `struct NODE` 结构体中，`number` 和 `name[20]` 的数据信息都存储在数据域中，相当于图 12.8 中的 A、B、C、D、E。`next` 是指针类型的成员，它指向 `struct NODE` 类型数据。用这种自己指向自己的数据类型的方法，就可以建立链表。链表的尾结点的 `next` 指针指向 `NULL`。

(2) 双向链表。当对单向链表进行操作时，若需要对某个结点的直接前驱进行操作，就必须从链表头开始查找。因为单向链表的指针域是指向直接后继的结点，因此就出现了既能存储直接后继结点地址的链域，又能存储直接前驱结点地址的链域，这就是一个双向链表。

在双向链表中，结点除含有数据域外，还有两个指针域。一个存储直接后继结点的地



址,称为右链域;另一个存储直接前驱结点的地址,称为左链域。

(3) 循环链表。循环链表与单向链表一样,都含有一个数据域和一个指向直接后继结点的指针域。唯一不同的是,循环链表没有尾结点。循环链表的最后一个结点的指针指向该循环链表的第一个结点,或者表头结点,从而构成一个环形的链。在建立一个循环链表时,还可以在最后一个结点后插入一个新的结点。判断循环链表是否到结尾,就是判断该结点链域的值是否为表头结点。当链域值等于表头指针时,说明已到结尾。



Note

专家点评

单链表是最简单的一种链表。但是,在查找结点时,只能查找后继结点。然而,双向链表可以查找前驱结点和后继结点,在循环链表中则可以查找任意位置的结点,因为循环链表是一个环形的链表,由任意结点都可以遍历到所有结点。

问题 237 如何在链表中的指定位置插入结点?

问题阐述

在链表中的指定位置插入一个结点,要求链表本身必须已按某种规律排好序。如何在链表中的指定位置插入新结点,需要掌握怎样找到插入的位置以及怎样实现插入?

专家解答

由于链表是链式结构的,因此要插入一个结点,就先要断开链表,才能将新的结点插入。就好比幼儿园的小朋友手拉手排成一队,要想将一个新来的小朋友插入队伍里,必须在需要插入的位置处将两个小朋友的手松开,然后让前面的小朋友的手拉住新来小朋友的手,新来小朋友的另一只手再拉住后面小朋友的手,这样就完成了插入,形成一个新的队列。

(1) 按照上面的思路,如果在第 i 个位置插入一个结点,需要找到插入的位置。

使用循环语句,从头结点移动指针 p ,开始遍历每一个结点,找到要插入位置的前一个结点。循环语句相应代码如下。

```
p=head;
while(p->next!=NULL)
{
    p=p->next;
    ++j;
}
```

跳出循环后,指针 p 就指向了 $i-1$ 结点的位置。

(2) 找到了插入的位置,就要实现插入操作。

将要插入的结点(即 s 指向的结点)指向链表中插入位置的下一个结点。相应代码



如下。

```
s->next=p->next;
```

将指针 p 指向的结点 (第 $i-1$ 个结点) 指向要插入的新结点 (s 指向的结点)。代码如下。

```
p->next=s;
```

插入新结点的过程如图 12.9 所示。

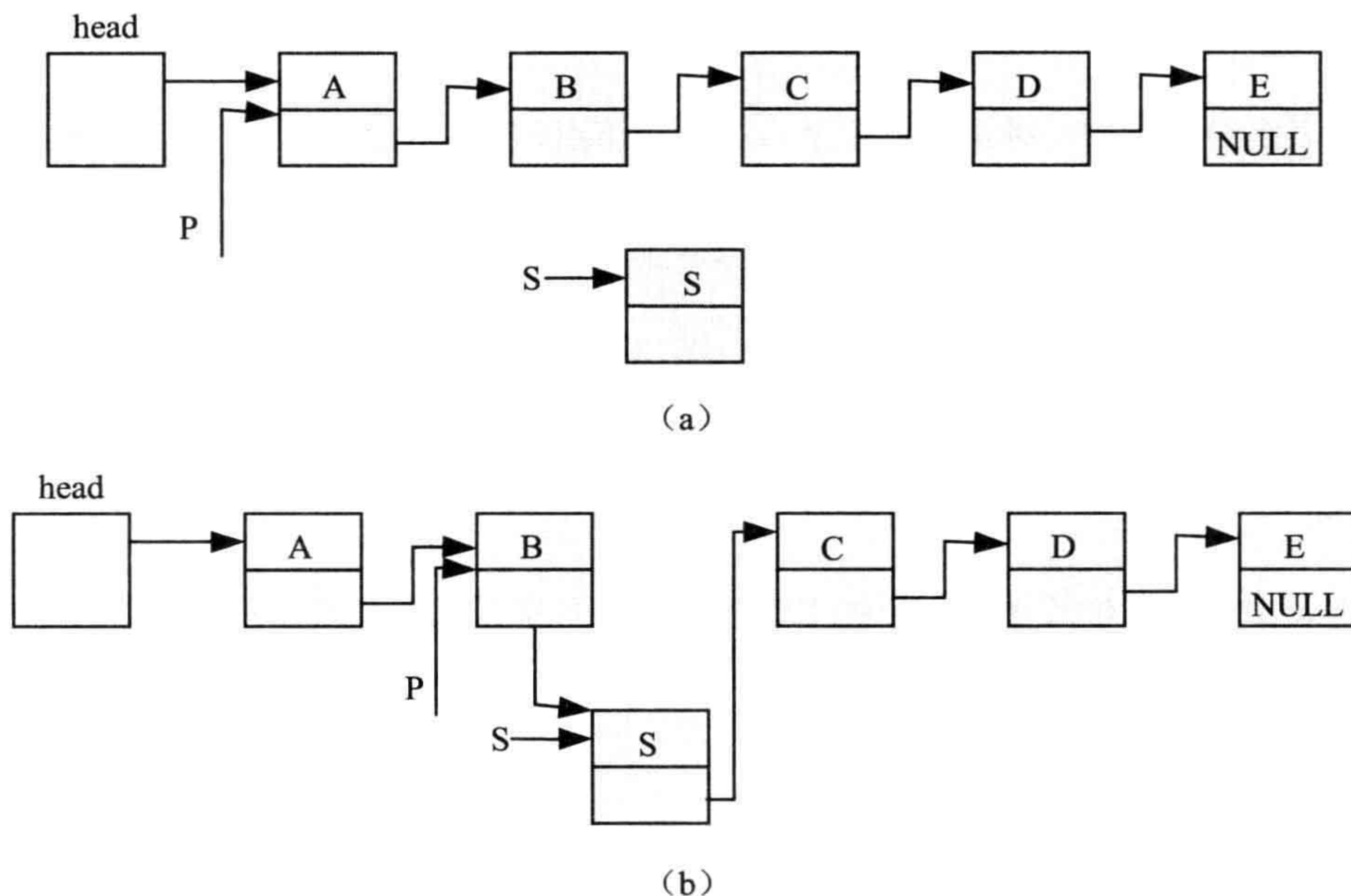


图 12.9 插入新结点的过程

专家点评

在插入新结点的过程中,必须先给需要插入的新结点的数据域赋值,然后将新结点的指针指向指针 p 所指的结点的直接后继,最后再将 p 结点的指针指向新生结点 s 。

问题 238 如何删除链表中指定位置的结点?

问题阐述

如何删除链表中指定位置的结点?

专家解答

删除链表中指定的结点,就像是排好队的小朋友手牵着手,将其中一个小朋友从队伍中分出来,只需将这个小朋友的双手从两边松开。



删除结点有两种情况:

(1) 被删除结点是头结点。这种情况使 head 指向第二个结点即可, 即 $\text{head}=\text{p}->\text{next}$;

(2) 被删除结点不是头结点, 这种情况只需将被删结点的前一结点指向被删结点的后一结点即可, 例如: $\text{pq}->\text{next}=\text{ph}->\text{next}$ 。

在删除了链表中的此结点后, 需要将此结点所占的内存释放掉, 以节省内存, 方便其他变量的应用。

删除链表结点的过程如下:

(1) 找到要删除的位置, 与插入结点时查找位置相同。相应代码如下。

```
p=head;
while(p&& j<i-1)
{
    p=p->next;
    ++j;
}
```

在遍历了结点并找到删除的位置后, 判断 p 所指向的后继结点是否为 NULL, 若后继结点为 NULL, 则说明不存在这个想要删除的结点, 程序直接退出。相应代码如下。

```
if(!p->next||j>i-1)
    exit(1);
```

(2) 找到要删除的结点的位置, 需要执行删除结点操作和释放内存操作。相应代码如下。

```
r=p->next;
p->next=r->next;
free(p);
```

实现删除指定位置的链表结点的过程如图 12.10 所示。

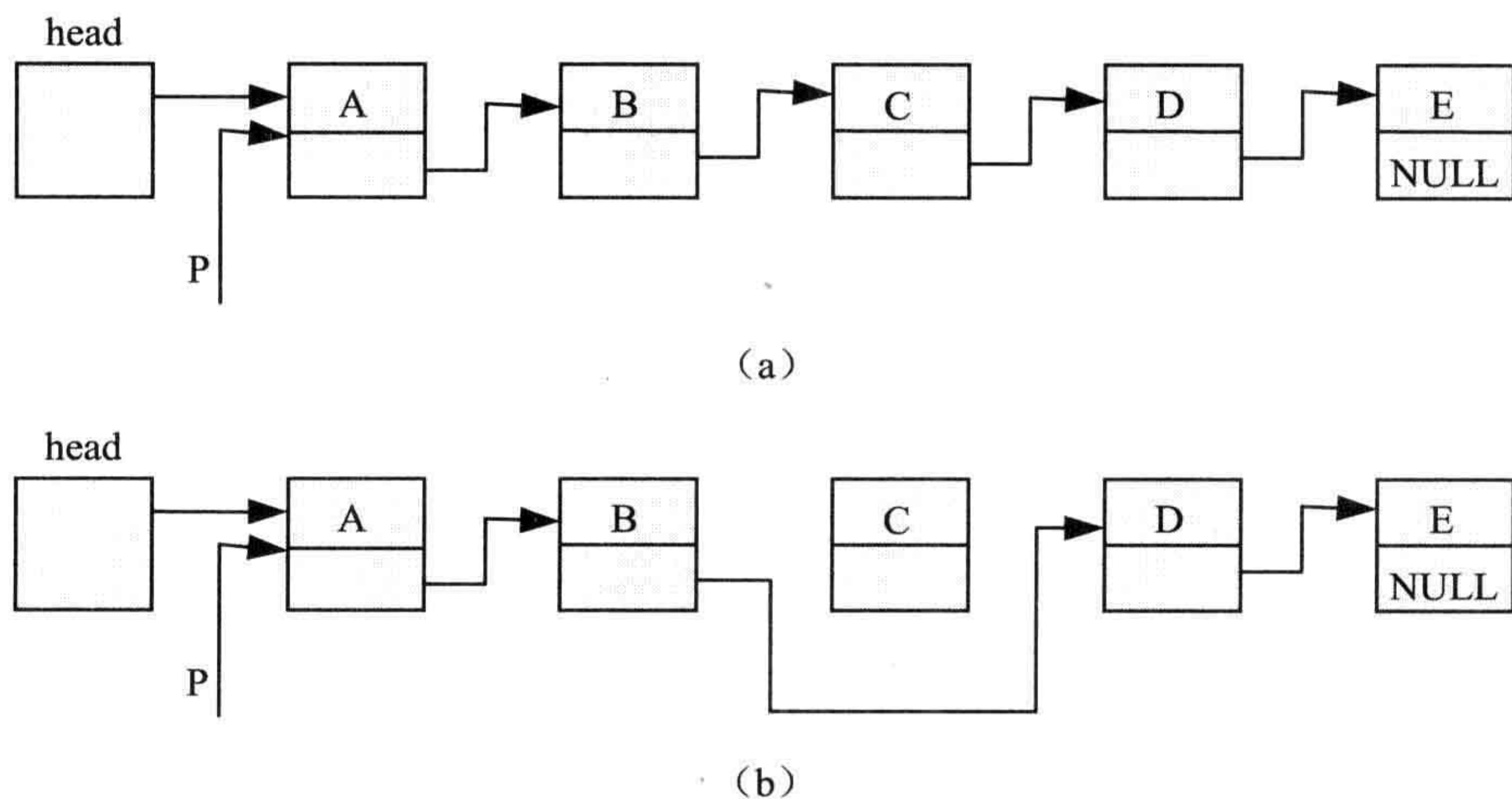


图 12.10 删除链表示意图





专家点评

在链表中删除结点，就是将结点从链表中分离出来，撤销原来的链接关系，而不是将结点从内存中抹掉。



Note

问题 239 如何创建一个动态链表？

问题阐述

创建动态链表就是指在程序执行过程中，从无到有，按照需求开辟结点和输入各结点数据，并建立起前后相连接的关系。那么，如何创建动态链表呢？

专家解答

以建立一个有任意名学生数据的单向动态链表为例，可以根据需要，动态地为学生分配内存，直到输入学号为 0，则结束输入，不再创建，然后将动态创建的学生信息输出。创建一个动态链表函数的相应代码如下。

```
stu *creat(void)                                /*创建动态链表函数*/
{
    stu *head,*p1,*p2;                          /*定义结构体类型的指针*/
    n=0;
    p1=p2=(stu *)malloc(LEN);                   /*开辟一个内存空间*/
    scanf("%d,%d,%f",&p1->num,&p1->age,&p1->score); /*输入结构体类型的数据*/
    head=NULL;                                  /*头指针置空*/
    while(p1->num!=0)                             /*判断学号输入是否为 0，若是 0 则跳出循环*/
    {
        n=n+1;
        if(n==1)head=p1;                       /*判断输入的是否为第 1 个数据信息，若是第一个
                                                数据信息，则将头指针指向 p1*/
        else
            p2->next=p1;                        /*将 p2 指向的下一个地址指向 p1*/
        p2=p1;                                  /*p2 指向 p1*/
        p1=(stu *)malloc(LEN);                 /*再次为 p1 开辟一个内存空间，存储下一个数据*/
        scanf("%d,%d,%f",&p1->num,&p1->age,&p1->score);
    }
    p2->next=NULL;                              /*p2 指向下一个地址指向的是空指针*/
    return(head);                              /*返回数据信息的头指针，以便从头输出*/
}
```

在创建函数中，若输入学号为 0，则说明链表建立完成。建立链表之前，首先使 head 指向一个空指针，表示此时的链表无结点，当建立了第一个结点后，就令 head 指向该结点。实现创建链表的过程如下：



(1) 在为 $p1$ 和 $p2$ 开辟了内存空间后,从键盘输入一个学生的数据信息给 $p1$ 所指的第一个结点。若学号不为 0,则这输入的是第一个数据,即 $n=1$,那么令 $head$ 指向该结点,即 $head=p1$,把 $p1$ 的值赋给 $head$,如图 12.11 所示。

(2) 建立完第一个结点后,再开辟一个结点并使 $p1$ 指向该结点。向结点中输入数据,如果第二个结点的学号不为 0,此时 $n=2$,则第二个结点就是第一个结点的直接后继。使 $p2$ 的指针域指向第二个结点,建立联系,此时链表的形式如图 12.12 所示。



Note

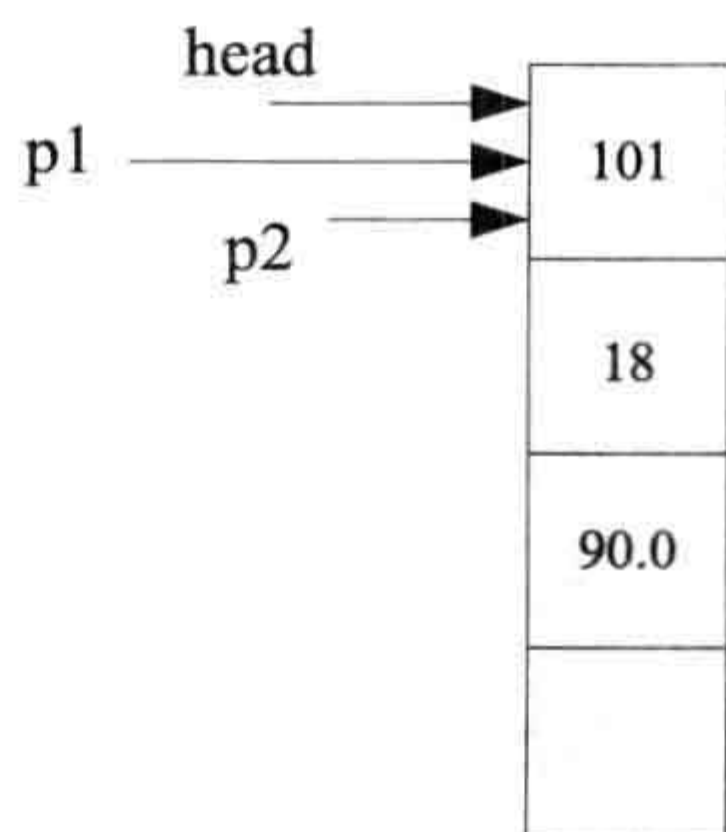


图 12.11 建立第一个结点

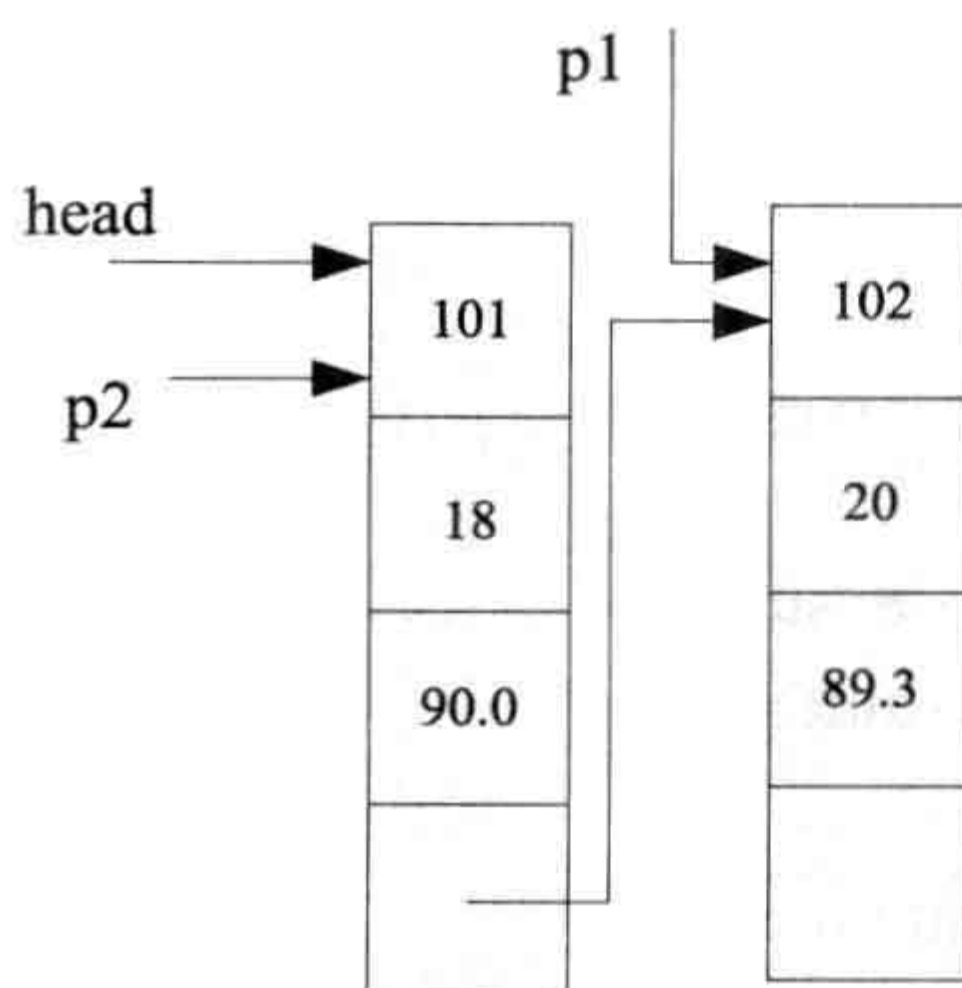


图 12.12 第二个结点的建立

(3) 第二个结点建立后,使 $p2$ 指向新建立的第二个结点,方便 $p1$ 再去开辟新的内存,建立下一个结点。建立下一个结点的方法同(2),第三个结点的建立如图 12.13 所示。

(4) 下一个结点的建立方法同(2),直到键盘输入学号为 0,则链表建立结束, `xuehaowei0` 这个结点不连接到链表中。

创建动态链表结束后,就可以在主函数中调用创建函数。该创建函数无传递参数,返回值为头指针。创建链表结束后,输出学生数据,运行结果如图 12.14 所示。

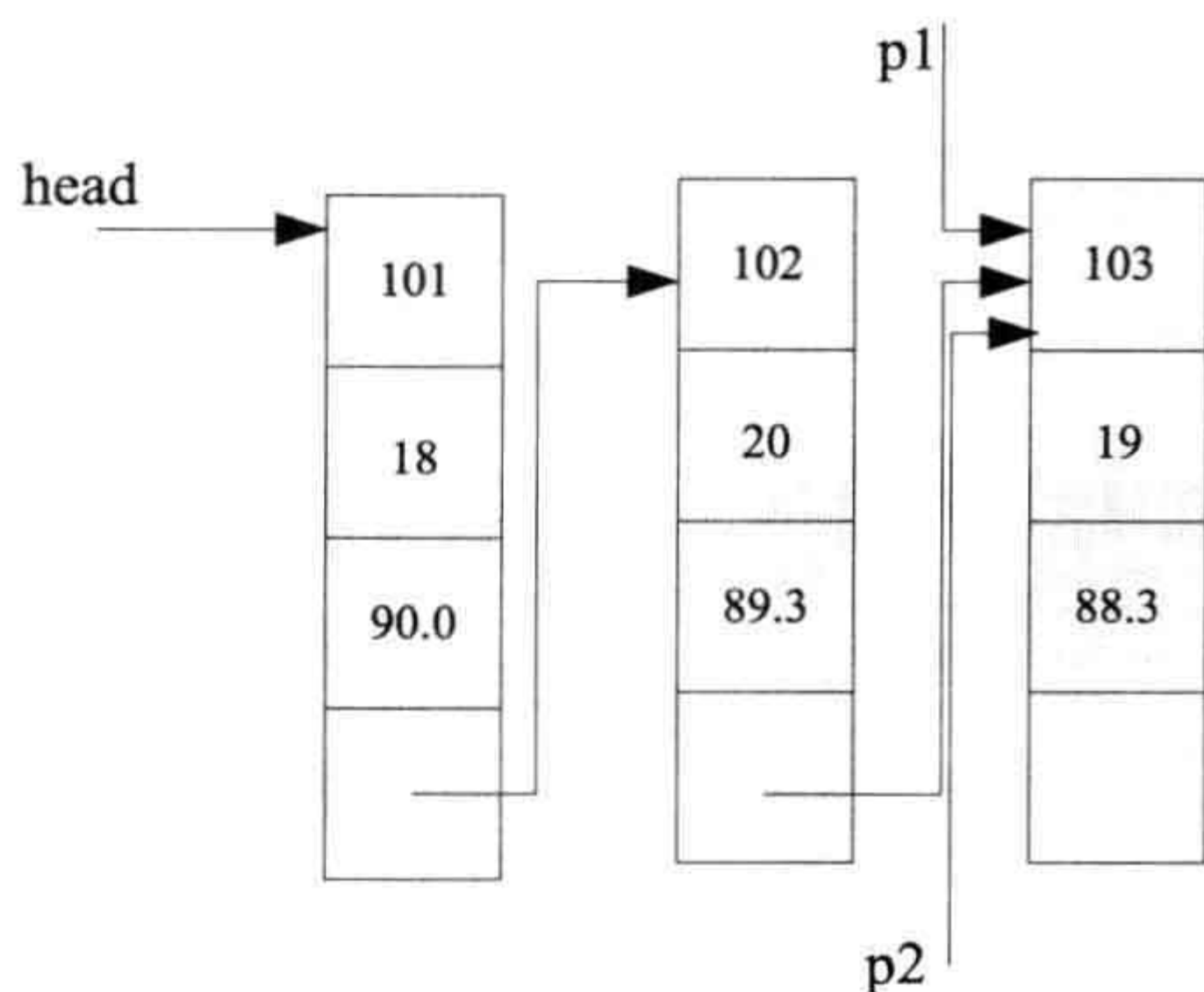


图 12.13 第三个结点的建立

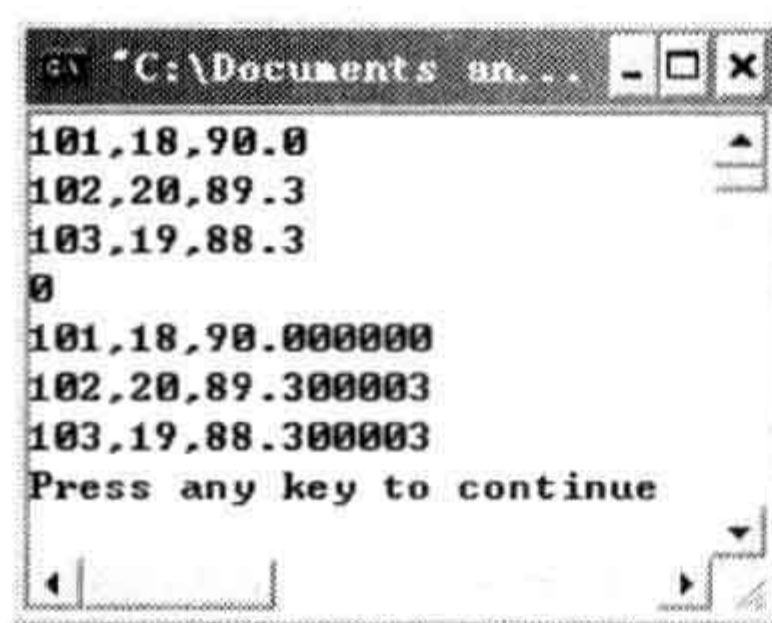


图 12.14 动态链表运行结果

专家点评

在创建动态链表的函数中,需要注意在开辟完一个结点后,需要先向结点中输入数据,再将前一个结点与后一个结点建立联系,即前一个结点的指针域指向后一个结点,这样才能将开辟的结点建立链式结构。



问题 240 指向结构体数组的指针如何应用?

问题阐述

在使用数组时,可以用指向数组或数组元素的指针和指针变量。同样,在结构体类型的数组及其元素中也可以应用指针或指针变量来指向。具体怎么应用呢?

专家解答

在如下代码中实现指向结构体数组的指针的应用。

```
#include<stdio.h>
struct student                                /*自定义的学生结构体类型*/
{
    int num;
    char name[20];
    int age;
    int score;
};
/*定义学生结构体数组*/
struct student stu[3]={101,"feifie",18,99},{102,"baibai",19,89},{103,"xixi",18,76}};
main()
{
    struct student *p;                        /*结构体类型的指针*/
    printf("学号\t姓名\t年龄\t平均成绩\t\n");
    for(p=stu;p<stu+3;p++)                    /*指向结构体类型数组的指针的应用*/
        printf("%5d%10s%4d%10d\n",p->num,p->name,p->age,p->score);
}
```

程序运行结果如图 12.15 所示。

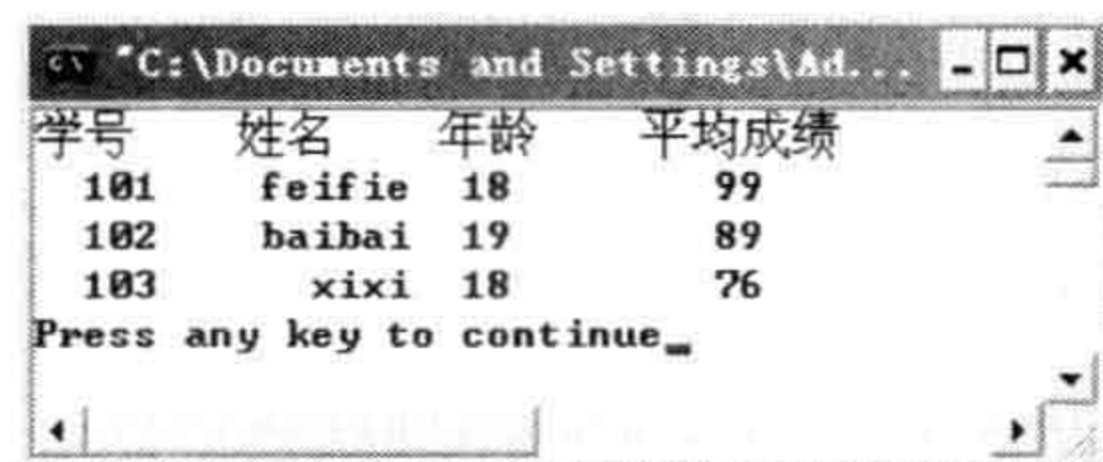


图 12.15 指向结构体数组的指针应用

在上述代码中, p 是指向 `struct student` 结构体类型的指针, `stu[3]` 是定义的结构体类型的数组, 数组内包含三个学生的信息, 在全局变量中为数组 `stu[3]` 赋初值。在 `for` 语句中, 先使 `stu` 数组的首地址赋给 p , 然后在数组范围内遍历 p , 每循环一次, p 自加 1。在执行了 `p++` 后, p 的值就等于 `stu+1`, 就是数组 `stu[1]` 的地址。在循环中执行 `p++`, 直到 p 指向的值变为 `stu+3` 处的值, 就不再执行循环体, 而跳出循环, 输出结束。



专家点评

在实现输出指向结构体数组的指针的应用中, 需要注意如果 p 的初值为 stu , 即数组的首地址, 则 $p+1$ 后 p 就指向下一个元素的起始地址。同时也要注意, 在程序中已经定义了 p 是一个指向结构体类型的指针变量, 不应用来指向 stu 数组元素中的某一成员, 也就不可以写成此形式: $p=stu[2].score$ 。



Note

第13章

位运算操作符

- ▶▶ 什么是位运算？位运算符包括哪些？
- ▶▶ 移位运算中如何补位？
- ▶▶ 移位运算符与加减运算符的优先级哪个较高？
- ▶▶ 什么是循环移位？
- ▶▶ 什么是位段，其优点是什么？
- ▶▶ 如何正确使用位段？
- ▶▶ 数在计算机中的存储单位有哪些？有几种存储形式？



问题 241 什么是位运算？位运算符包括哪些？

问题阐述

数学中的加、减、乘、除四则运算，恐怕谁都不会陌生。那么在 C 语言中，什么是位运算？位运算符包括哪些呢？

专家解答

1. 位运算

所谓位运算，就是对二进制位进行运算。位运算不是以字节为单位进行的，而是对内存中存储数据的每个二进制位进行的运算。

每个二进制位都是由 0 和 1 组成的，通常最右端称为低位，左端称为高位。正确、熟练地进行位运算，可以帮助编写复杂的程序，节省内存空间。

在运算的过程中，必不可少的是运算操作符，如加、减、乘、除等。那么位运算的操作符包括哪些呢？

2. 位运算符

C 语言提供了 6 种位运算符，分别是按位与 (&)、按位或 (|)、按位异或 (^)、按位取反 (~)、左移 (<<) 和右移 (>>)。除了按位取反运算符为单目运算符外，其余都是双目运算符。

(1) 按位与 (&)

将两个参与运算的数据表示为二进制位的形式，按二进制位进行“与”运算。即如果两个相应的二进制位都是“1”，则运算结果为“1”否则都为“0”。例如 6&9，即 6 按位与 9，计算过程可以表示如下。

$$\begin{array}{r}
 00000110 \text{ (6)} \\
 (\&) \quad 00001001 \text{ (9)} \\
 \hline
 00000000 \text{ (0)}
 \end{array}$$

注意：

如果参与&运算的是负数，则以补码形式将此数表示为二进制数，然后按位进行“与”运算。

(2) 按位或 (|)

按位或也是两个二进制位进行运算，如果两个相应的二进制位有一个是“1”，结果就是“1”，否则结果为“0”。例如 6|9，计算过程可以表示如下。

$$\begin{array}{r}
 00000110 \text{ (6)} \\
 (|) \quad 00001001 \text{ (9)} \\
 \hline
 00001111 \text{ (15)}
 \end{array}$$





(3) 按位异或 (^)。

异或运算的含义是参与运算的两个二进制位同号，则结果为“0”，不同则结果为“1”。例如 $6 \wedge 7$ ，计算过程可以表示如下。

$$\begin{array}{r} 00000110 \text{ (6)} \\ (^) \quad 00000111 \text{ (7)} \\ \hline 00000001 \text{ (1)} \end{array}$$

(4) 按位取反 (~)。

按位取反是位运算符中唯一一个单目运算符，用来对一个二进制数按位取反，即“0”变“1”，“1”变“0”。例如 ~ 9 ，计算过程可以表示如下。

$$\sim 00001001 \text{ 得到 } 11110110$$

(5) 左移 (<<)。

“<<”用来将一个数的各二进制位左移若干位。例如 $10 << 3$ ，将 10 表示成二进制数为 00001010，左移 3 位，右补 0，得到 01010000，转换为十进制数为 80。

(6) 右移 (>>)。

“>>”用来将一个数的各二进制位右移若干位。移到右端的低位被舍弃；对无符号数，高位补 0。例如 $33 >> 2$ ，将 33 表示成二进制数为 00100001，右移 2 位得到 00001000，转换成十进制数为 8。

专家点评

对于位运算，上面只是简单回答了什么是位运算，以及位运算的几种运算形式。其实，在位运算中还存在一些较为复杂的运算，如负数的位运算，以及不同长度的两个数据之间的位运算等。用到时要注意补位。

问题 242 移位运算中如何补位？

问题阐述

移位运算包括左移和右移，其功能是将参与运算的数据的各二进制位全部左移或者右移若干位。运算符“<<”和“>>”都是双目运算符，运算符左侧为需要移位的数据，右侧为需要移动的位数。那么在移位过程中，如何补位呢？

专家解答

左移运算的功能是将“<<”左边的运算数的各二进制位全部左移若干位，由“<<”右边的数指定移动的位数。移位后，高位丢弃，低位补 0。

例如 $a=35$ ，表示成二进制数形式为 00100011。那么 $a << 3$ ，就是将 a 的二进制数的各个二进制位都向左移 3 位，则移出的高 3 位舍弃，即 001 舍弃，而低三位就会空出，用 0 补位，结果为 00011000，转换成十进制数为 24。





右移运算的功能是将“>>”左边的运算数的各二进制位全部右移若干位，由“>>”右边的数指定移动的位数。移位后，同样是高位丢弃，低位补 0。

例如 $a=9$ ，表示成二进制数为 00001001。那么 $a>>1$ ，就是将 a 的二进制数的各个二进制位都向右移 1 位，则移出的低 1 位舍弃，而高一位空出，用 0 补位，结果为 00000100，转换成十进制数为 4。

在移位运算中应该注意的是，对于有符号数，在右移时符号位也随之移动，当运算数为正数时，最高位补 0；当运算数为负数时，符号位为 1，不同的编译系统补位不同，可能补 1 也可能补 0。在 Turbo C 和很多系统中规定高位补 1。

专家点评

补位是移位运算中的重点，要记住左移与右移的补位位置，以及正负数是补 0 还是补 1。若不了解编译系统在负数右移时高位是补 0 还是 1，可以编写一个简单的移位程序来证明。

问题 243 移位运算符与加减运算符的优先级哪个较高？

问题阐述

运算符都有优先级，优先级高的先运算，优先级低的后运算；若优先级相同，则根据结合方向，从左到右或者从右到左运算。那么移位运算符与加减运算符的优先级哪个较高呢？

专家解答

首先通过一个例子验证一下，它们的优先级哪个较高。代码如下。

```
main()
{
    printf("a=%d\n",0x01<<2+3);
    printf("b=%d\n",0x01<<3-2);
}
```

如果左移位运算符“<<”的优先级别高于加减运算符，那么 a 的值应该按照结合方向从左到右运算，先将十六进制数 0x01 左移 2 位，然后将左移后的结果转换为十进制数加上 3，结果为 7。同理， b 的运算结果应该为 6。那么程序的运行结果是否就是这样的呢？不是的，而是 $a=32$ ， $b=2$ ，如图 13.1 所示。

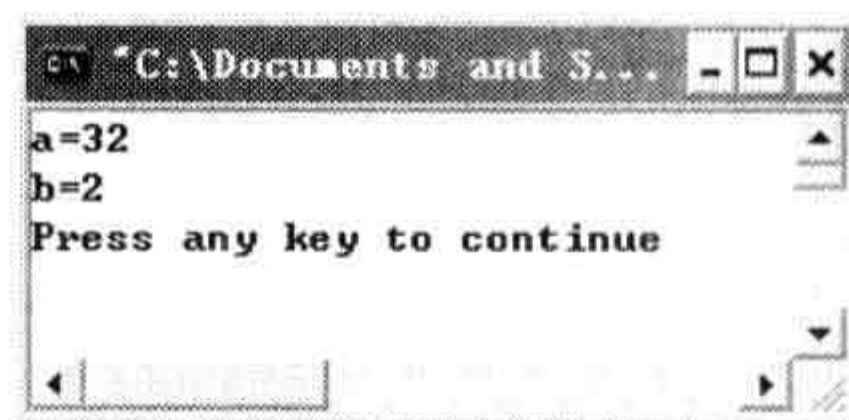


图 13.1 移位运算

这样的结果是怎样算出来的呢？原因就是加减运算符的优先级别高于移位运算符。 a 的运算过程为：先进行 $2+3$ 运算，得到 5，然后再将 0x01 左移 5 位，得到 32。 b 的运算过程如 a ，先进行 $3-2$ 运算得到 1，然后将 0x01 左移 1 位，得





到结果为 2。

那么在 32 位系统下, $0x01 \ll 2+30$ 或 $0x01 \ll 2-3$ 这两个表达式的结果会是多少呢?

一个整型数长度为 32 位, 左移 32 位会怎样呢? 答案是: 溢出。那么左移 -1 位呢? 同样也会溢出。由此可以看出, 左移和右移的长度是有规定的, 不能大于数据的长度, 也不能小于 0。

专家点评

在进行移位运算时, 要注意移动的位数, 不要超过范围, 以免造成溢出。

问题 244 什么是循环移位?

问题阐述

左移位是将二进制位的数据向左移动, 移动位数由移位运算符右侧的十进制整数决定。向右移位与向左移位相似, 只是方向不同。那什么是循环移位?

专家解答

循环移位就是在移位的过程中, 不丢失移位前原操作数的位, 而是将移出的二进制位数据写入需要补位的位置。循环移位的结果如图 13.2 所示。

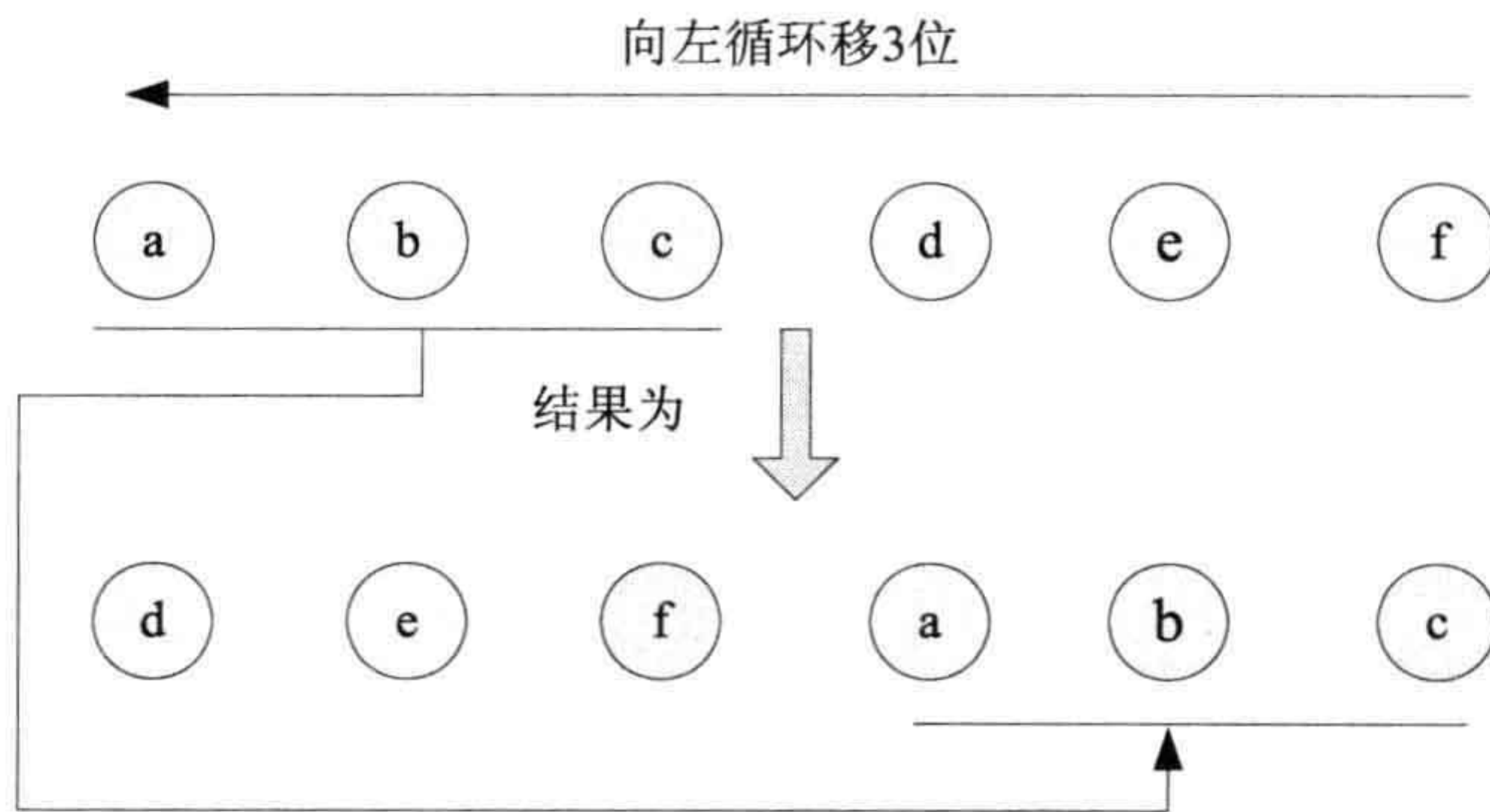


图 13.2 循环左移

下面举例说明循环移位的过程。例如, 要求将十六进制数 a 进行左循环移位。为实现循环左移, 有如下算法步骤。

(1) 将 a 的左端 n 位先放到 b 中的低 n 位。可以用如下语句实现。

```
b=a>>(16-n);
```

(2) 将 a 左移 n 位, 其左面低 n 位补 0。可以用如下语句实现。

```
c=a<<n;
```




(3) 将 c 与 b 进行按位或运算。即:

```
c=c|b;
```

这样就实现了循环左移。循环右移可以采用同样的思想。

说明:

在进行移位运算时, 采用十六进制数更能清晰地观察出变化。



Note

用十六进制数实现循环左移的实例程序如下。

```
main()
{
    unsigned a,b,c,n,z;
    scanf("a=%x,n=%d",&a,&n);          /*键盘输入十六进制数 a 和移动位数 n*/
    b=a>>(16-n);                        /*将高 n 位变成低 n 位*/
    printf("b=%x\n",b);
    c=a<<n;                             /*将 a 整体左移 n 位, 低 n 位补 0*/
    printf("c=%x\n",c);
    z=c|b;                               /*或运算, 各个位上的数保持不变*/
    printf("%x\n%x\n",a,z);
}
```

程序运行结果如图 13.3 所示。

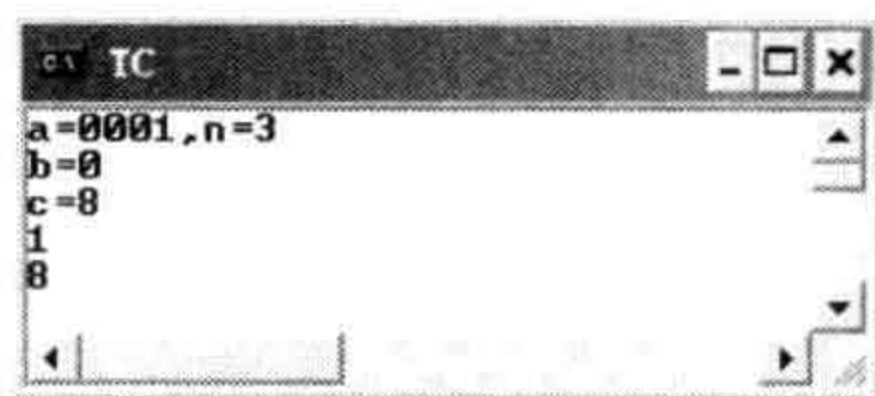


图 13.3 循环左移

专家点评

在循环移位的算法步骤中, 重要的是要理解 (3) 的或运算, 按位或运算可以使得到的 z 保持 b 的高位和 c 的低位均不变。

问题 245 什么是位段? 其优点是什么?

问题阐述

在内存中, $1\text{byte}=8\text{bit}$, 即 1 字节等于 8 位。位由两个值组成, 即 0 和 1。因此, 存储在计算机中的 1 字节, 可以看成是 8 个二进制数字 (0 和 1) 组成的串。了解了内存空间的最小单位, 那什么是位段? 位段又有什么优点呢?



专家解答

1. 什么是位段

在 C 语言中，允许在一个结构体中以位为单位来指定其成员所占内存的长度。这种以位为单位的成员，便称为“位段”或者“位域”。例如：

```
struct data_bit
{
    unsigned a:3;
    unsigned b:3;
    unsigned c:4;
    unsigned d:8;
    int n;
}data1;
```

其中 a、b、c、d 分别占 3 位、3 位、4 位、8 位，一共占 2 字节；而 n 为 int 型，占 2 字节，在这个结构体中总共占了 4 字节。

2. 位段的优点

使用位段的一大优点就是可以节省不必要的空间。实际上，在数据通信、电子应用、参数检测等领域中，控制信息往往只占一个字节中的几个二进制位。例如，开关的设计，只需要一个二进制位就可以表示，0 表示关，1 表示开，没有必要使用一个字节。那么，位段就避免了这种资源的浪费，可以根据实际应用中的需要合理分配内存。

专家点评

关于位段的定义，有一点要注意，位段成员的类型必须指定为 unsigned 或 int 类型。

问题 246 如何正确使用位段？

问题阐述

位段的出现是为了更加合理地分配内存，避免内存空间的浪费，那么如何正确使用位段呢？

专家解答

在介绍如何正确使用位段之前，先来看一个例子。

例如，需要定义 4 个状态 a1、a2、a3、a4，4 个状态互不相同，那么通常都会定义为：

```
char a1, a2, a3, a4;
```

总共占了 4 个字节的内存空间。其实，只需要两个二进制位就可以表示，如，00,01,10,11；



可以定义为：

```
struct pact_data
{
    unsigned int a1:2;
    unsigned int a2:2;
    unsigned int a3:2;
    unsigned int a4:2;
};
```

这种位段的定义方法，只占了 8 位，即 1 字节。

对位段中数据的引用方法与结构体引用成员变量的方法相同。例如：

```
struct pact_data data;
data.a1=0;
data.a2=1;
data.a3=2;
data.a4=3;
```

在使用位段赋值时，需要注意位段允许的最大范围。如果是两个二进制位，那么范围是 0~3。如果赋值时超出了位段成员的范围，如 `data.a4=4`，这样就错了，因为上述例子中的最大值为 3。但是在这种情况下，系统会自动取这个数的低位赋予这个成员变量。

在使用位段时，还会遇到以下情况。

(1) 某一位段要从另一个字节开始存放，可以使用如下方式。

```
unsigned a :2;
unsigned b :3;
unsigned   :0;
unsigned c :3;
```

其中第三条语句用了长度为 0 的位段，表示一个位段从一个存储单元开始存放。

(2) 定义无名位段。例如：

```
unsigned a :1;
unsigned b :2;
unsigned   :3;
unsigned c :4;
```

其中第三条语句定义了无名位段，表示这 3 个位不能使用。

(3) 在使用位段时，可以使用整型格式符、长整型格式符、八进制格式符、十六进制格式符等多种格式符输出。

(4) 位段同样可以在表达式中引用。进行加、减、乘、除等运算时，系统会自动地将结果转换为整型数。

专家点评

以上关于位段的使用都是正确的，但需要注意的是，在使用位段时，位段的长度不要



Note



大于存储单元的长度，并且不能像正常的结构体那样定义位段数组。同时，一个位段必须存储在同一个存储单元中，不能跨两个单元。



Note

问题 247 数据在计算机中的存储单位有哪些？ 有几种存储形式？

问题阐述

在一个房间中容纳多少人，以“个”为单位来计算；一个书柜里有多少书，以“本”为单位计算。那么一台计算机中存储的数据是以什么单位来计算呢？有几种存储形式？

专家解答

1. 计算机中的数据存储单位

在计算机中以位（bit）、字节（byte）和字（word）为单位存储数据。

（1）位（bit）

位又称为比特，是计算机存储数据的最小单位。一个二进制数中的每一个数值 0 或 1，均代表 1 位（或称 1 比特）。

（2）字节（byte）

一个字节为 8 位二进制数，是存储数据的基本单位。通常情况下，一个英文字母或符号用 1 字节来存放，一个汉字用 2 字节来存放。正如长度单位除了有厘米，还有毫米、米、千米等，字节也是，存储容量单位还有千字节（KB）、兆字节（MB）、吉字节（GB）。这些单位之间存在以下换算关系：

$$1\text{B}=8\text{ bit (位)}$$

$$1\text{KB}=1024\text{B}$$

$$1\text{MB}=1024\text{KB}$$

$$1\text{GB}=1024\text{MB}$$

（3）字（word）

字是由若干个字节组成的一个单元，一个字可以存放一个数据或指令。字长与机器有关。通常，286 机中，一个字由 2 字节组成；486 机中，一个字由 4 字节组成。

2. 数据的存储形式

数据在计算机中有 3 种编码形式，分别介绍如下。

（1）原码。原码是该数绝对值的二进制表示。例如，9 的原码为 0000 0000 0000 1001。那么 -9 的原码是什么呢？在原码中，最高位称为“符号位”，即用最高位表示数的正负，正数最高位用 0 表示，负数最高位用 1 表示。因此，-9 的原码为 1000 0000 0000 1001。

（2）反码。一个正数的反码与其原码相同。例如，+11 的原码是 0000 0000 0000 1011，其反码与原码相同，也是 0000 0000 0000 1011。一个负数的反码是符号位（即最高位）不变，其余各位取反。例如，-11 的原码为 1000 0000 0000 1011，其反码为 1111 1111 1111 0100。



(3) 补码。正数的补码与原码相同，负数的补码为反码加 1。例如：

+11 的补码为：0000 0000 0000 1011。

-11 的补码为：反码加 1，即 1111 1111 1111 0100 加上 1，结果为 1111 1111 1111 0101。

专家点评

有了补码，在进行减法运算时，就可以理解成补码相加来计算了。



Note

第14章

存储管理

- ▶▶ 与内存息息相关的重要概念有哪些？
- ▶▶ 指针指向不合法引起了哪些内存问题？
- ▶▶ 内存分配与释放引起的常见问题有哪些？
- ▶▶ 什么是内存越界？什么是内存泄露？各是如何产生的？
- ▶▶ C语言提供了哪些动态内存分配函数？
- ▶▶ malloc()函数与 calloc()函数有什么区别？
- ▶▶ 内存耗尽怎么办？
- ▶▶ 动态内存会被自动释放吗？
- ▶▶ 高位优先与低位优先的不同之处是什么？
- ▶▶ free()和 delete()怎样处理指针？
- ▶▶ 怎样利用好敏感的内存资源？



问题 248 与内存息息相关的重要概念有哪些?

问题阐述

C 语言、C++语言和 C#语言, 这三门语言, 一个比一个加号 (+) 多, C 语言没有加号, C++有两个加号, C#有四个加号。随着语言的发展, 一个比一个简单, 很多问题系统都给做了, 无需程序员考虑。然而, 最基层的也是最重要的, C 语言却在很多地方都需要程序员自己动手, 如内存管理。在管理内存中, 还要时刻考虑内存的泄露等问题。下面介绍一下与内存息息相关的几个重要概念。

专家解答

1. 野指针

从字面上理解, “野”字就好像是没有人管、行为粗鲁、不守规矩的意思。例如, 野猪就是在森林里没有人管、靠自己生存、没有束缚、行为粗鲁的猪, 并不像家养的猪那么干净、守规矩等。

野指针也是这样的, 没有规矩。野指针不同于 NULL 指针, 它是指向“垃圾”内存的指针。

野指针的成因可能会有如下几种情况:

(1) 指针变量没有被初始化。指针变量在创建的同时应该被初始化, 指向 NULL 指针或者指向一块合法的内存, 否则它的指向是随机的。

(2) 指针的操作超过了变量的作用范围。

(3) 指针被释放或者删除后, 没有被置为 NULL, 在以后的程序中被误认为是合法的。

2. 栈 (stack)

栈是用来保存局部变量, 栈上的内容只在函数的范围内存在, 函数运行结束, 这些内容也会被销毁。栈的特点就是效率高, 但空间大小有限。

3. 堆 (heap)

堆是由 malloc()、calloc()等函数或者 new 操作符获得的内存, 由 free()函数和 delete()函数释放内存。若在程序中没有应用 free()函数或者 delete()函数进行释放操作, 则内存会一直占用, 直到程序结束。堆的特点是使用灵活, 空间比较大。

4. 静态区

静态区用于保存自动全局变量和 static 变量。静态区的内容在整个程序中都存在, 由编译器在编译的时候分配内存。

专家点评

内存的分配管理十分重要, 管理不当就会为程序带来重大隐患。一般来说, 内存可以



Note



理解为栈、堆和静态区三部分。

问题 249 指针指向不合法引起了哪些内存问题?

问题阐述

通常在定义了指针变量后,应该对其初始化,使其指向一块合法的内存,否则很有可能造成野指针,使程序瘫痪。下面举例介绍一下由指针指向不合法造成的几个内存问题。

专家解答

在自定义的一个结构体变量 `person` 中,由一个字符型的指针变量作为成员变量,在引用该成员变量时出现错误,相应错误代码如下。

```
struct person
{
    int id;
    char *name;
    int age;
}per,*p1;
int main()
{
    strcpy(per.name,"feifei");
    per.id=2201;
    per.age=18;
    return 0;
}
```

很多人在看这段代码时并没有发现什么错误,但需要注意的是,在这个结构体内部定义了 `char *name` 成员时,只是给它分配了四个字节,并没有使这个空间指向一个合法的地址。因此,在使用字符串复制函数向这个成员变量所指的内存上复制字符串时出错,因为这块内存无权访问。需要在使用这块内存之前,为成员变量 `name` 使用 `malloc()` 函数以获得一块合法的内存。

即使为结构体类型的指针变量 `p1` 使用 `malloc()` 分配了一块内存,仍不可以对 `p1->name` 进行复制字符串操作。原因同上,并没有为指针类型的成员变量 `name` 分配内存。

专家点评

很多指针变量被隐藏到结构体中,人们就忽略了为它们分配内存。然而,在为结构体类型的指针变量分配内存时,一定要注意分配足够大的内存,若分配内存不足,如 `sizeof(struct person*)`,则会导致指针成员变量 `name` 没有被分配到内存,以致出现上述的错误。



问题 250 内存分配与释放引起的常见问题有哪些?

问题阐述

对于内存的管理,为了节省空间,在使用过分配的内存后要及时释放。下面介绍一下常见的几种内存分配和释放引起的问题。

专家解答

1. 为指针分配内存太小

为指针分配内存,要注意内存的大小,如下代码造成了内存大小不够,出现越界错误。

```
char *p1="feifei";  
char *p2=(char*)malloc(sizeof(char)*strlen(p1));  
strcpy(p2,p1);
```

已知字符串常量具有字符串结束符“\0”,所以占用七个字节的内存。然而,在计算字符串 p1 的长度时,结果却是六个字节的长度。这是因为在 malloc() 一块内存时,分配的内存是六个字节。所以在字符串复制函数中将 p1 指向的内容复制给 p2 会出现越界。因此,分配内存时应该再加上一个字节。

2. 成功分配了内存,同时要初始化

编写程序要有一个初始化的概念。例如,在定义一个变量时,首先要做的就是初始化,但是往往在刚开始定义这个变量时,并不确定这个变量该赋什么初值,这时最好是为变量赋 0 或者 NULL。不同的数据类型的变量,赋予 0 的形式不同,例如:

- (1) 整型变量 int i=0;
- (2) 浮点型变量 float=0.0;
- (3) 指针变量 char *p=NULL;
- (4) 数组变量 int a[10]={0}。

3. 内存已被释放,但继续使用

(1) 使用 free(p) 语句释放了指针 p 指向的内存,但是继续通过 p 来访问内存。这是错误的,释放后需要将 p 置 NULL。

(2) 在不清楚内存中栈上变量的生命周期时,会出现在函数中返回指向该函数中数组的指针。

(3) 分配内存太复杂、太多,不明确释放的是哪一块内存,就会造成再次对已经释放的指针继续使用。

专家点评

编写程序代码要养成良好的习惯。在定义了变量的同时要为其初始化,其余变量不初



Note



始化可能引起一些小问题，但是指针变量未被初始化可能出现野指针。在释放了内存后没有把这个指针置 NULL，那么这个指针就成了野指针。



Note

问题 251 什么是内存越界？什么是内存泄露？ 二者是如何产生的？

问题阐述

什么是内存越界？什么是内存泄露？分别是如何造成的？

专家解答

1. 内存越界

内存越界就是成功地分配了内存，并且已经初始化，但是在操作过程中越过了内存的边界。应用如下代码对内存越界进行说明。

```
char *p1="mingrikeji";  
char*p2=(char *)malloc(sizeof(char)*strlen(p1));  
strcpy(p2,p1);
```

字符串 p1 占用 11 个字节，其中包括最后的结束标识符“\0”。但是在为 p2 分配内存空间时，用 strlen(p1) 得到的字符串 p1 的长度为 10 个字节。因此，将 p1 中字符串赋予 p2 时出现了越界。

说明：

内存越界问题通常出现在数组和指针的操作上。

2. 内存泄露

如果在分配了内存空间并使用完后，没有及时地释放掉指针所占用的内存空间，而再次使用这个指针时又为其分配内存。如此操作，有限的内存空间就会被消耗，逐渐减少，这种现象称之为内存泄露。涉及释放内存的就是使用 malloc() 等函数分配的内存，因此会产生泄露的内存是堆上的内存。

专家点评

定义了一个指针后，要及时为这个指针分配内存空间，这是为了防止野指针的出现。在不使用这块内存后，要及时释放掉该内存，这不止是为了节省空间，更重要的是为了防止内存泄露。



问题 252 C 语言提供了哪些动态内存分配函数?

问题阐述

C 语言中提供了几种相关函数, 动态地开辟内存和释放存储单元。下面介绍一下这几种函数的功能。

专家解答

(1) malloc()函数。

函数原型为:

```
void *malloc(unsigned int size);
```

该函数的功能是分配长度为 size 字节的内存块。

如果分配成功, 则返回指向被分配内存的指针; 否则返回空指针 NULL。注意, 当内存不再使用时, 要用 free()函数释放内存块。

(2) calloc()函数。

函数原型为:

```
void *calloc(unsigned n,unsigned size);
```

该函数的功能是在内存的动态区存储中分配 n 个长度为 size 的内存块。

如果分配成功, 则返回指向被分配内存的指针; 否则返回空指针 NULL。同样, 当内存不再使用时, 要用 free()函数释放内存块。

同时, 用 calloc()函数可以为二维数组开辟动态存储空间, n 为数组元素个数, 每个元素长度为 size。

(3) realloc()函数。

函数原型为:

```
void *realloc(void *mem_address,unsigned int newsize);
```

该函数的功能是改变 mem_address 所指内存区域的大小为 newsize 长度。

如果重新分配内存成功, 则返回指向被分配内存的指针; 否则返回空指针 NULL。当内存不再使用时, 要应用 free()函数将内存空间释放。

(4) free()函数。

函数原型为:

```
void *free(void *p);
```

在 malloc()函数、calloc()函数和 realloc()函数中均提到在不使用内存空间时要应用此函数释放内存空间, 因此该函数的功能就是释放指针 p 所指向的内存空间。如果 p 为 NULL 或者指向不存在的内存块, 则不做任何操作。



专家点评

注意，free()函数所释放的内存空间一定要是 malloc()函数、calloc()函数或者 realloc()函数所分配的内存。



Note

问题 253 malloc()函数与 calloc()函数有什么区别？

问题阐述

malloc()函数与 calloc()函数都是用来开辟内存空间的，那么它们有什么区别呢？

专家解答

malloc()函数包含在 stdlib.h 头文件中，作用是在内存中动态地分配一块内存。因为 malloc()函数分配的内存空间是在堆中，而不是在栈中，所以在使用完这块内存后一定要使用 free()函数将其释放掉。

calloc()函数也包含在 stdlib.h 头文件中，使用该函数分配一个整型数组内存空间，代码如下：

```
int *p;  
p=(int *)calloc(2,sizeof(int));
```

在上面的代码中，p 为一个整型指针，使用 calloc()函数分配内存数组，第一个参数表示分配数组中元素的个数，而第二个参数表示元素的类型，最后将返回的指针赋给 p 指针变量，p 的指针就是该数组的首地址。

这两个函数的区别之一就是参数上的区别，malloc()分配一块 size 大小的内存，而 calloc()分配一个 n*size 大小的内存块；区别之二就是返回内存块的状态不同，malloc()分配的内存块没有被清零，而 calloc()分配的内存块是清了零的。

专家点评

虽然这两个函数的目的是相同的，都是开辟一块内存空间，但是仍然各有千秋，各不相同。在使用的时候，要根据程序员自己的需要，选择开辟内存空间的函数。

问题 254 内存耗尽怎么办？

问题阐述

内存耗尽，顾名思义，就是有限的内存不够用了。那么，内存耗尽了怎么办呢？



专家解答

如果申请动态内存时找不到足够大的内存块，`malloc()`函数和 `new()`函数将返回 `NULL` 指针，宣告内存失败。

解决内存耗尽可以使用如下三种方法：

(1) 判断指针是否为 `NULL`。如果是，则马上使用 `return` 语句终止本函数。例如：

```
void func(void)
{
    C *p=new C;
    if(p==NULL)
    {
        return;
    }
    ...
}
```

说明：

此例为 C++ 中的例子，用于说明内存耗尽的解决方法。其中，A 为类名，如 “`class C {...};`”。

(2) 判断指针是否为 `NULL`。如果是，则马上用 `exit(1)` 终止整个程序的运行。例如：

```
void func(void)
{
    C *p=new C;
    if(p==NULL)
    {
        exit(1);
    }
    ...
}
```

(3) 为 `new()` 函数和 `malloc()` 函数设置异常处理函数。

通常，如果一个函数内有多处需要申请动态内存，就会使用 (2) 的解决方法来处理。

专家点评

在出现内存耗尽时，一定要采取解决措施。如果让系统自行解决，而不用 `exit(1)` 终止程序，很有可能导致计算机死机。



Note



问题 255 动态内存会被自动释放吗?

问题阐述

当指针灭亡了, 动态内存会随着被自动释放吗?

专家解答

一个函数体内的局部变量是存放在栈中的, 栈的作用范围仅限于这个函数。函数结束, 这个局部变量就会被收回, 自动灭亡。那么, 下面这个例子的指针 `p` 的动态内存会自动灭亡吗?

```
void func(void)
{
    char *p=(char*)malloc(256);
}
```

这个函数中的 `p` 确实是自动灭亡了, 但是动态分配的内存却依然存在, 没有被收回。

指针指向一个 `malloc` 分配的动态内存, 这个指针看上去有些“叛逆”。指针随着函数的结束而被收回, 然而内存却依然存在。所以内存被释放并不代表指针灭亡了或者为 `NULL` 了。

因此, 分配内存容易, 释放内存难。正如, 花钱容易, 挣钱难。

专家点评

当程序终止运行时, 虽然指针被销毁, 内存被收回, 但是不可以草率结束, 为了程序的可移植性, 要给指针设置为 `NULL`。

问题 256 高位优先与低位优先的不同之处是什么?

问题阐述

C 语言的最大特色就是可移植性好。根据机器类型的不同, 高位优先与低位优先也不同。那么, 最好的可移植的 C 程序应该同时适用这两种类型的计算机。下面了解一下高位优先与低位优先的不同之处。

专家解答

所谓的高位优先, 就是最低的地址放在高位字节上, 而低位优先就是最低的地址放在低位字节上。通过图 14.1 形象地理解一下低位优先与高位优先的不同。

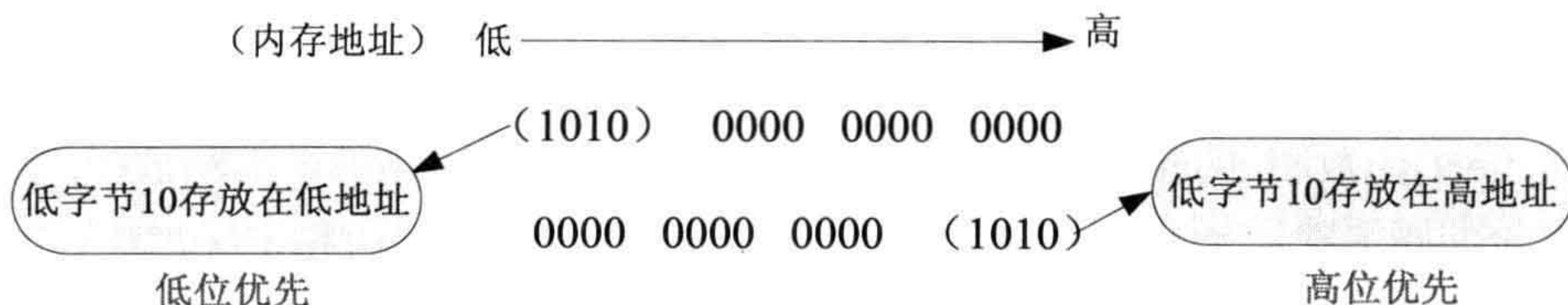


图 14.1 高位优先与低位优先

若机器为低位优先，则将低位字节 10 存放在低地址中，机器读取出来的结果为 10；如果机器为高位优先，则将低位字节 10 存放在高地址中，机器读取出来的结果为 0。

可以通过如下程序来证明机器是高位优先还是低位优先。代码如下。

```
#include<stdio.h>
main()
{
    int a=10;
    short b;
    memcpy(&b,&a,2);/*从 a 中拷贝 2 个字节放到 b 中*/
    printf("%d\n",b);
}
```

运行结果如图 14.2 所示。

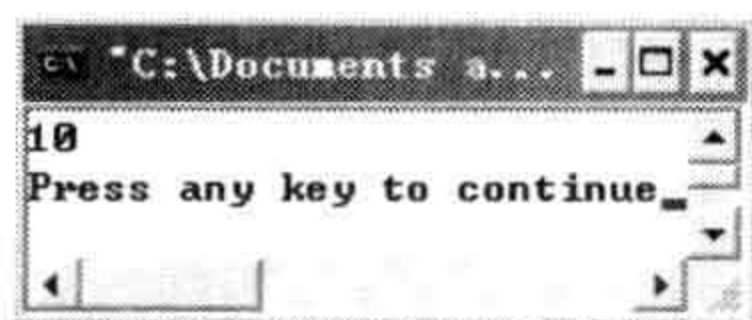


图 14.2 验证机器是高位优先还是低位优先

由运行结果为 10 可以得知，运行此程序的机器为低位优先。

在计算机中，对于 int、short、long 数据类型存在字节顺序，而 char 类型只有 1B，因此与字节顺序无关。对于 float 和 double 类型的值，没有一种标准的存储模式。

专家点评

高位优先与低位优先的区别仅仅在于机器是喜欢从左向右数，还是喜欢从右向左数。

问题 257 free()和 delete()怎样处理指针？

问题阐述

从字面上理解，free()是释放，给予自由的意思；而 delete()的含义比较直接，是删除的意思。这两个函数的目的就是不想再与这个指针有任何关联，那么它们到底怎样处理了不再使用的指针呢？





专家解答

其实 `free()` 函数和 `delete()` 函数起到的作用只是将该指针所指的内存释放掉，但并没有把指针本身彻底消除。

观察如下程序，通过调试器跟踪查看 `p` 指针的地址的变化。注意 `free()` 释放该指针后，地址是否为 `NULL`。相应代码如下。

```
#include<stdio.h>
main()
{
    char *p=(char*)malloc(100);
    strcpy(p,"mingrikeji");
    free(p);
    if(p!=NULL)
    {
        strcpy(p,"hello mr");
    }
}
```

当程序运行完语句“`strcpy(p,"mingrikeji");`”后，可以观察到调试器中的 `p` 指针的地址和存储的内容，结果如图 14.3 所示。

当执行完 `free(p)` 后，即释放指针 `p` 结束，`p` 的地址是否指向 `NULL` 呢？观察图 14.4 所示的跟踪调试（2）。

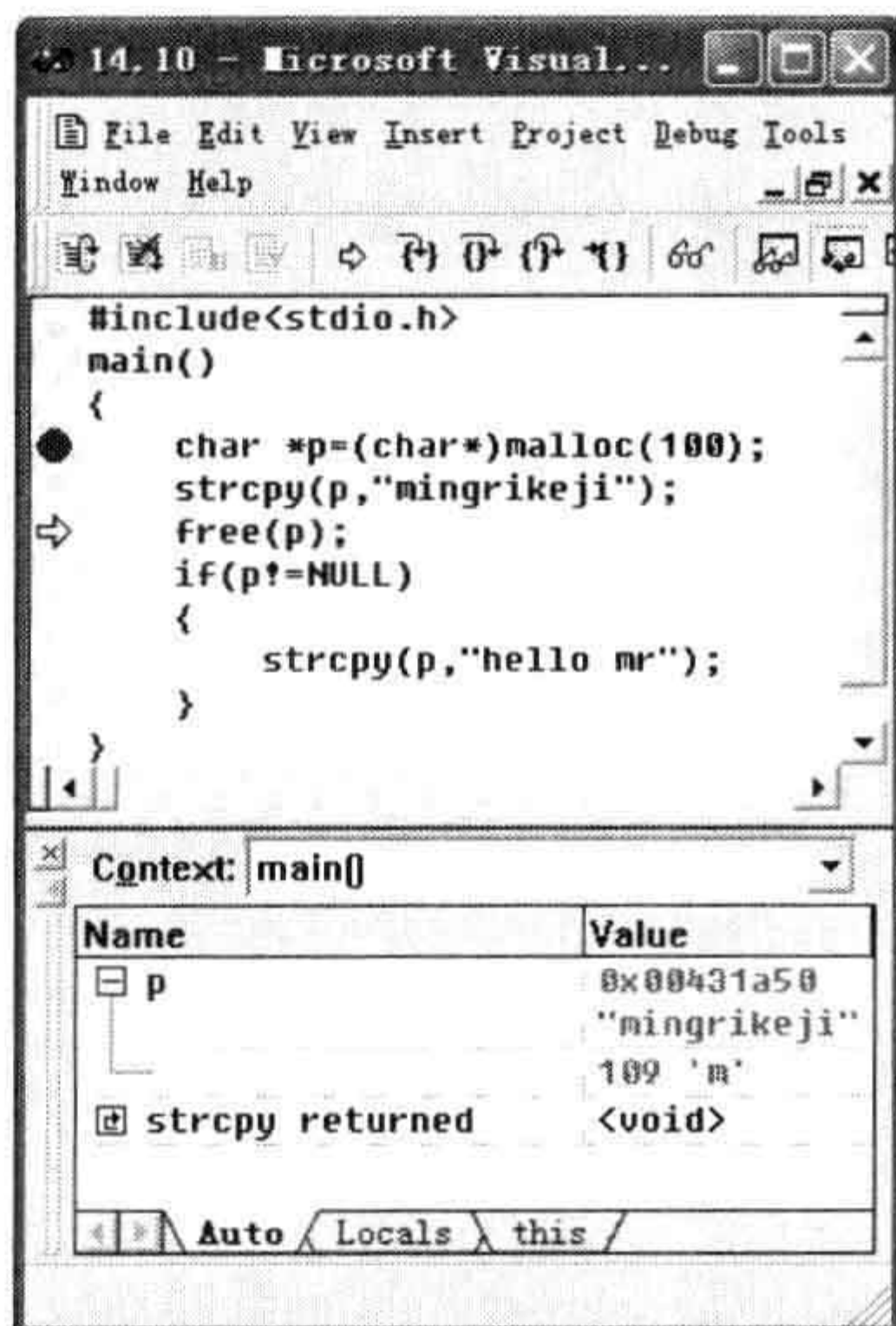


图 14.3 跟踪调试（1）

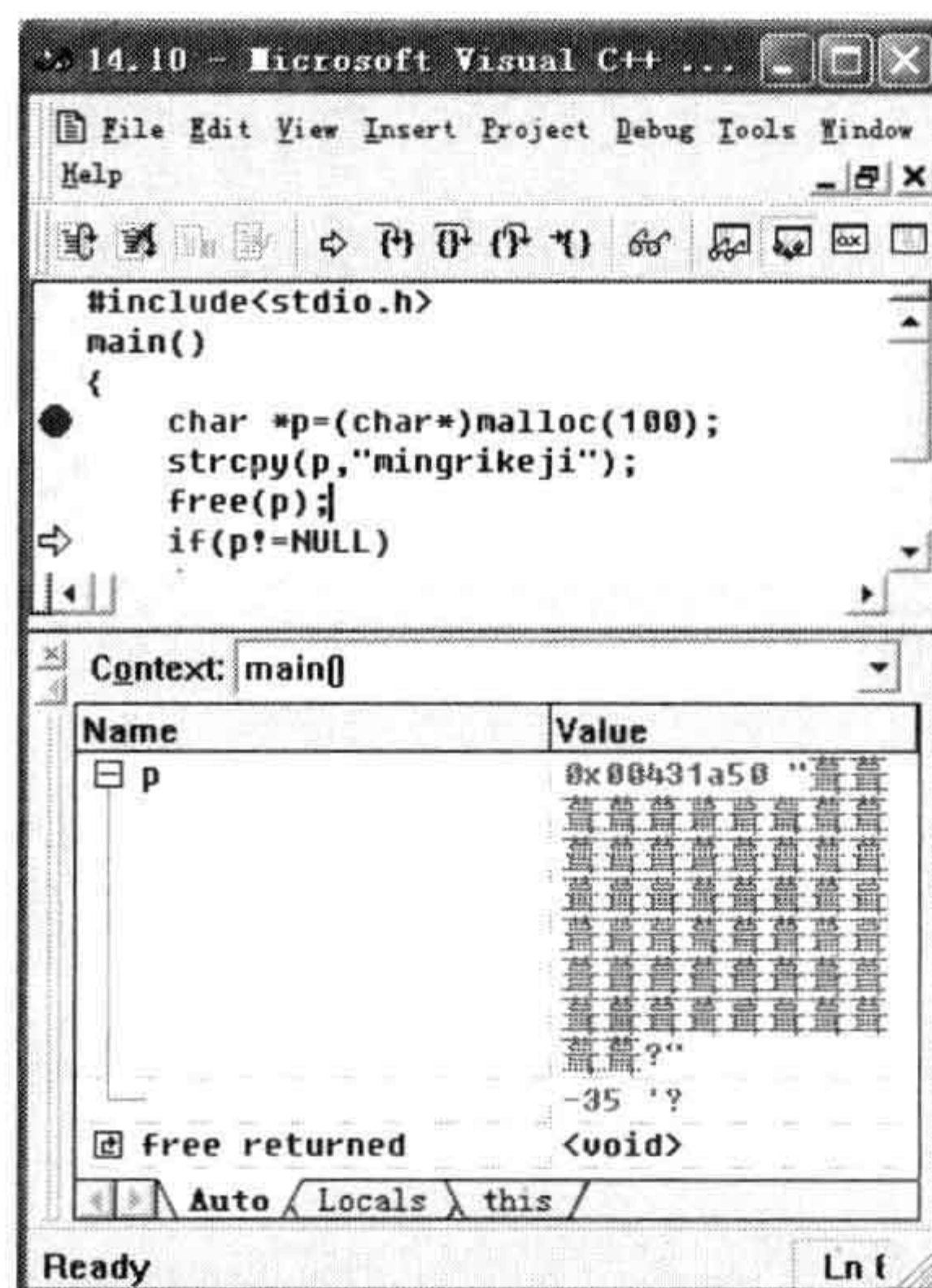


图 14.4 跟踪调试（2）

通过跟踪调试的图 14.3 和图 14.4 可以发现，指针 `p` 被 `free` 释放以后，其地址仍然不变，并不是 `NULL`，只是该地址中存放的内容由“`mingrikeji`”变为了垃圾内容。此时，指针 `p` 就成为了“野指针”，如果不把 `p` 置 `NULL`，会让人误以为 `p` 是一个合法的指针，而继续使用。通常，在继续使用之前，会通过语句 `if (p != NULL)` 进行防错处理，但是在



此程序中这条语句并没有起到防错作用，因为此时的 `p` 并没有指向合法的内存块。

专家点评

为了不让指针 `p` 成为“野指针”，而造成不必要的麻烦，在 `free()` 释放掉指针后，要将指针 `p` 设置为 `NULL`。



Note

问题 258 怎样利用好敏感的内存资源？

问题阐述

内存是一个很敏感的资源，就像敏感肌肤一样，要百般呵护。同时，内存资源又很复杂，如何正确地使用内存资源不容易，能够正确并且合理地利用好内存资源就更不容易。下面通过一个例子，来讲解怎样利用好这个敏感的内存资源？

专家解答

定义一个函数 `itoa()`，实现的功能是将整数转换为字符串。在主函数中调用该功能函数，观察输出结果，相应代码如下。

```
#include<stdio.h>
char *itoa(int i)
{
    char buffer[20];
    sprintf(buffer,"%d",i);
    return buffer;
}
main()
{
    char *str1=itoa(7);
    char *str2="hello";
    printf("str1=%s\nstr2=%s\n",str1,str2);
}
```

程序输出的结果会是 `str1=7`，`str2=hello` 吗？

`buffer` 是一个定义在 `itoa()` 函数中的局部变量，它的内存空间位于栈中，其作用范围也仅限于这个 `itoa()` 函数中。当退出 `itoa()` 函数时，`buffer` 中的内容将被收回。此时，这块内存可能存放其他内容，因此 `buffer` 这个局部变量返回给主函数是达不到想要的结果的。于是，输出的 `str1` 中的值是不确定的，并且不可能是“7”。程序的运行结果如图 14.5 所示。

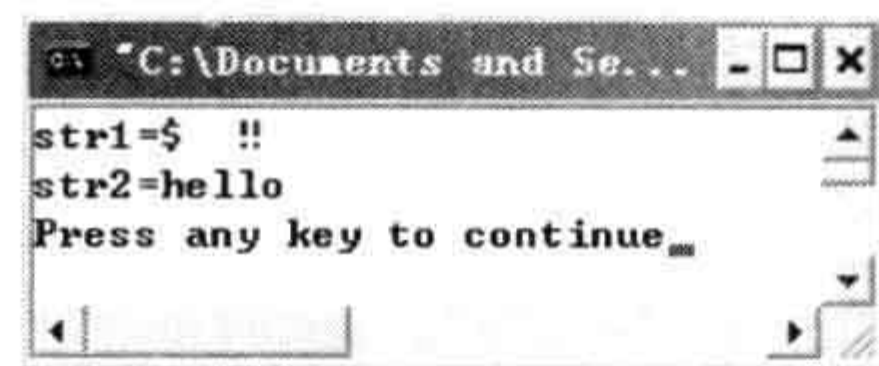


图 14.5 内存资源的使用



那么，如何解决这个内存问题呢？

首先，了解几个概念：程序中有不同的内存段、静态区、栈和堆，存放在栈中的数据会在调用函数结束后收回，那么存放在静态区和堆中的呢？已知在静态区中定义的变量，在整个软件执行过程中都有效，并且在堆中由程序显示分配和收回，如果不收回，就是内存泄露。

由此概念，可以有以下两种解决方案：

(1) 在 `itoa()` 函数中定义一个静态变量，如 `static char buffer[20]`，这样静态变量 `buffer` 在整个程序中都有效。但是，静态变量有一个弊端，不能保证相同的输入一定会有相同的输出。

(2) 还可以将该变量存储在堆中。使用 `malloc()` 函数为 `buffer` 分配一块内存，在不使用该变量时再释放该内存，否则会引起内存泄露。

其实，最好的解决方法是自己使用的内存由自己申请和释放。上述解决方案 (2) 在函数中自己分配空间，但是要调用者释放该函数，容易遗漏，造成内存泄露。因此，自己的事情自己做可以避免错误。

解决方案为：在定义函数时，将 `buffer` 做为传递参数，由调用者申请和释放 `buffer` 的空间。`itoa()` 函数只负责将转换的结果存放到 `buffer` 中。

专家点评

对于内存资源的使用，要了解不同内存段起到的作用，并且最好是自己申请的内存自己释放，这样可以减少内存的错误使用。

第15章

预处理和函数类型

- ▶▶ 在头文件中`#if`、`_STDC_`等字符起什么作用?
- ▶▶ 如何书写多条语句宏?
- ▶▶ 预处理中`#`和`##`运算符是什么意思?
- ▶▶ 一个头文件可以包含另一个头文件吗?
- ▶▶ `#include<>`和`#include""`有什么区别?
- ▶▶ 什么是无参宏定义?
- ▶▶ 什么是带参宏定义?
- ▶▶ 怎样写参数个数可变的宏?
- ▶▶ `#pragma`预处理的作用是什么?
- ▶▶ 条件编译的表达形式有哪些?
- ▶▶ 如何应用内部函数?
- ▶▶ 如何应用外部函数?



问题 259 在头文件中 #if、_STDC_ 等字符起什么作用？

问题阐述

通常，一些程序员都不会去研究头文件中的内容是什么含义，总觉得乱乱的，有很多 #if、_STDC_、#line 等字符，那么这些字符都各代表什么呢，在头文件中又起到什么作用呢？

专家解答

在头文件中存在类似于 #if、#undef、#error 等样式的字符，这些都是 ANSI 标准定义的 C 语言预处理指令。相关的预处理指令有：

- (1) #define：作用是宏定义。
- (2) #undef：作用是撤销已定义过的宏名。
- (3) #include：作用是使编译程序将另一源文件嵌入带有 #include 的源文件中。
- (4) #if、#else、#elif 和 #endif：其中，#if 的一般含义是如果 #if 后面的常量表达式为 true，则编译它与 #endif 之间的代码，否则跳过这些代码。命令 #endif 标识一个 #if 块的结束。#else 命令的功能有点像 C 语言中的 else，当 #if 失败，就进入 #else 的另一个选择。#elif 命令意义与 else if 相同，它形成一个 if else-if 形式的语句，可以进行多种编译选择。
- (5) #ifdef 和 #ifndef：其中，预处理指令 #ifdef 表示“如果有定义”，而 #ifndef 表示“如果无定义”，两个指令是条件编译的另一种方法。
- (6) #line：它的作用是改变当前行数和文件名称，它们是在编译程序中预定义的标识符命令的基本形式，如 #line member[“filename”]。
- (7) #error：作用是在编译程序时，只要遇到 #error 就会生成一个编译错误的提示消息，并停止编译。
- (8) #pragma：该指令是为实现时定义的命令，它允许向编译程序传送各种指令。例如，编译程序可能有一种选择，它支持对程序执行的跟踪，可用 #pragma 语句指定一个跟踪选择。

在头文件中，除了这些预处理指令，还有一些由下划线组成的字符，它们又起什么作用呢？在 ANSI 标准 C 语言中，还定义了由两个下划线和一个标识符组成的宏。例如：

- (1) _LINE_：表示正在编译的文件的行号。
- (2) _FILE_：表示正在编译的文件的名字。
- (3) _DATE_：表示编译时刻的日期字符串，如 “12 Dec 2011”。
- (4) _TIME_：表示编译时刻的时间字符串，如 “15:17:29”。
- (5) _STDC_：判断该文件是不是定义成标准 C 程序。

专家点评

若编译器是非标准的，那么可能仅支持以上宏的一部分，也可能还提供其他的预定义



的宏名。

问题 260 如何书写多条语句宏？

问题阐述

宏定义是为了将复杂的、经常用到的字符串用一个简单的字符表示，这样在修改字符串时可以只在宏定义处修改一次，避免了多次修改程序中的复杂字符串而引起错误。那么，一个经常用到的并且很复杂的多条语句可以用宏定义吗？如何书写呢？

专家解答

宏定义是可以用来定义多个语句的。在宏调用时，把这些语句又代换到源程序中，例如：

```
#define AB(a1,b1,a2,b2) a1=m*t;b1=m*(t+5);a2=m*h;b2=m*(h+3);
main()
{
    int m=2,t=12,h=5;
    int A1,B1,A2,B2;
    AB(A1,B1,A2,B2);
    printf("A1=%d\nB1=%d\nA2=%d\nB2=%d\n",A1,B1,A2,B2);
}
```

程序的运行结果如图 15.1 所示。

以上为宏定义多条语句，宏名 AB 代表着四个赋值语句，其中四个形参 a1、b1、a2、b2 分别为赋值语句的赋值符号的左边的变量。在主函数中进行宏调用时，把四个语句展开，并用主函数中提供的实参代替形参，把计算结果送入实参中。这么多条语句，如果一行放不下，该怎样处理好呢？

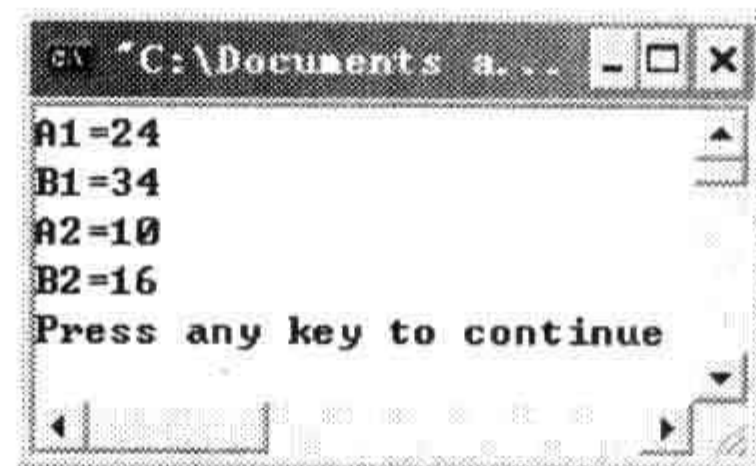


图 15.1 多条语句宏定义

如果我们在一行代码的行尾放置一个反斜杠，C 语言编译器就会忽略行尾的换行符，而把下一行的内容也算作是本行的内容，这里反斜杠可以起到续行的作用。反斜杠还有一个作用，就是定义跨多行的宏。例如，宏体中的字符串太长，可以采用反斜杠“\”分隔开，如果不使用反斜杠，直接用回车在下一行书写，编译会出错。书写多行宏语句，正确的书写方法如下。

```
#define PRINT printf("abcdefghijklmnpqrst \
    uvwxyz");printf("hello world! "); \
printf("game over...");
```

专家点评

如果一个 C 程序语句较长，都写在一行中有时会编译出错，提示信息过长，此时可以



Note



考虑使用反斜杠续行。

问题 261 预处理中#和##运算符是什么意思?

问题阐述

有人认为,在 C 语言中使用“#”运算符的就是预处理,是不是呢?“##”又是什
么呢?

专家解答

在程序中,最为常见的是#define 宏定义指令,下面通过这个指令理解一下“#”的作
用。编写一个预处理指令,代码如下。

```
#define S(x) printf("x 的平方为%d\n",(x)*(x))
main()
{
    S(4);
}
```

看到这个程序,很容易理解程序运行结果是什么。程序的运行结果如图 15.2 所示。

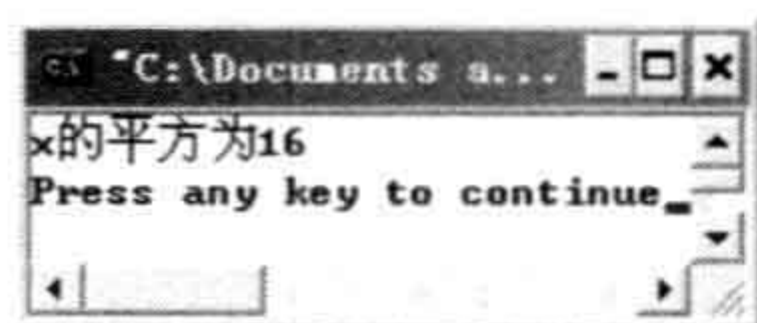


图 15.2 宏定义指令演示 (1)

那么,怎样才能使输出时的字符 x 被当做需要替换的符号呢?希望能够在输出的字
符串中包含宏参数,如下代码就可以满足这个要求。

```
#define S(x) printf("#x的平方为%d\n",(x)*(x))
main()
{
    S(4);
}
```

程序运行结果如图 15.3 所示。

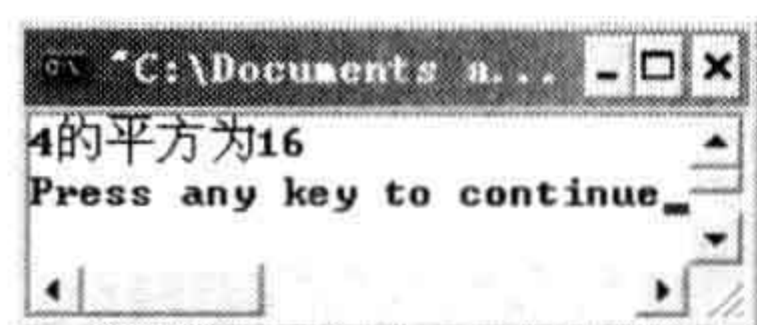


图 15.3 宏定义指令演示 (2)

结果是令人满意的,那么“#”运算符在预处理指令中的作用也就清晰可见了。它的
功能是可以把语言符号转化为字符串。

“##”运算符与“#”运算符一样,可以用于宏函数的替换部分。这个运算符还可以



起到粘合剂的作用，即将两个语言符号组合成单个语言符号。例如：

```
#define X(n) n##n
main()
{
    printf("%d\n",X(3));
}
```

可以得到 `n` 被宏函数替换成 3，并且粘合到一起，得到 33。程序运行结果如图 15.4 所示。

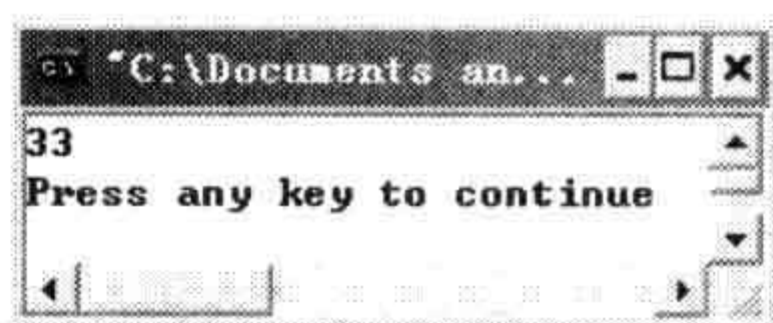


图 15.4 ##运算符的应用

专家点评

C 语言博大精深，小小的运算符“#”和“##”就有如此大的功能，但是它们的功能却很少有人了解。

问题 262 一个头文件可以包含另一个头文件吗？

问题阐述

通常用在文件头部的被包含文件称为“标题文件”或“头部文件”，简称“头文件”。头文件常以“.h”为后缀，表示 head（头）的意思。那么，这个头文件可以包含另一个头文件吗，即嵌套包含文件？

专家解答

头文件中往往存放一些宏定义的常量、函数原型以及结构体类型定义和全局变量等。其中，头文件在程序中被使用，需要一个 `#include` 命令包含这个头文件，并且，一个 `#include` 命令只能指定一个被包含文件。关于头文件，在一个被包含文件中又可以包含另一个被包含文件。也就是说，文件包含是可以嵌套的。

很多人认为“嵌套包含文件”应该避免，因为这样会让相关的定义更加难以找到。如果一个文件被包含两次，就会导致重复定义的错误，同时会令 `makefile` 的人工维护十分困难。另外，它使模块化使用头文件成为一种可能，即一个头文件可以包含所需要的一切，而不是让每个源文件都包含需要的头文件。类似 `grep` 的工具（或 `tags` 文件）使搜索定义十分容易，无论它在哪里。

有一种比较流行的头文件定义技巧，例如：

```
#ifndef HFILENAME_USED
```



Note



```
#define HFILENAME_USED
```

```
...
```

```
#endif
```

```
/*头文件内容*/
```

每一个头文件都使用了一个独一无二的宏名。这令头文件可自我识别，以便可以安全地多次包含，而自动 makefile 维护工具可以很容易地处理嵌套包含文件的依赖问题。

专家点评

由于头文件特殊的宏名，使一个头文件可以包含另一个头文件，避免了嵌套引发的很多重复定义的错误。

问题 263 #include<>和#include“”有什么区别？

问题阐述

有两种头文件包含的形式，一种是用尖括号将头文件括起，一种是用双引号将文件括起。那么，这两种形式有什么区别呢？

专家解答

这两种包含头文件的形式都是合法的，也是经常在代码中看到的，两者的区别在于“<>”语法通常用于标准或系统提供的头文件；而双引号“”通常用于程序自己的头文件。

如果在头文件用“<>”括起，那么在用到头文件中的常量或者函数时，需要系统到存放 C 库函数头文件所在的目录中寻找要包含的文件，这称之为标准方式，当在系统提供的文件中找不到该头文件时，就会再次搜索程序自己的头文件。

若到头文件用双引号“”括起，系统先在用户当前目录中寻找要包含的文件，若找不到，再按标准方式查找。

了解了两种包含方式的区别，那么在写程序的时候，为了节省查找时间，需要在调用库函数时用“<>”包含相关的头文件。如果包含用户自己编写的头文件，最好用双引号“”。

注意：

对于双引号形式包含头文件，如果文件不在当前目录下，可以在双引号内给出文件的路径。

专家点评

掌握了常见的两种包含文件的方式，可以进一步增加对 C 语言的认识。同时，在自己写代码的时候，可以根据不同的情况选择文件包含方式，以节省机器查找文件的时间。



问题 264 什么是无参宏定义?

问题阐述

宏定义是由源程序中的宏定义命令完成的。宏代换是由预处理程序自动完成的。在 C 语言中,宏定义分为无参和有参两种。那么,什么是无参宏定义呢?

专家解答

宏定义就是用一个指定的标识符来代表一个字符串,无参宏定义就是不带参数的宏定义,它的一般形式可以表示如下。

`#define 标识符 字符串`

其中,“#”表示一条预处理命令,凡是以“#”开头的均为预处理命令;“define”为宏定义命令;“标识符”为所定义的宏名;“字符串”是需要替换的代码,可以是常数、表达式、格式串等。

例如,宏定义一个表达式和一个格式串,然后在主函数中代换,进行运算,输出结果。相应代码如下。

```
#define A x*x+2/y
#define P printf("%f\n",z)
main()
{
    int x=2,y=4;
    double z=A+3;
    P;
}
```

程序的运行结果如图 15.5 所示。

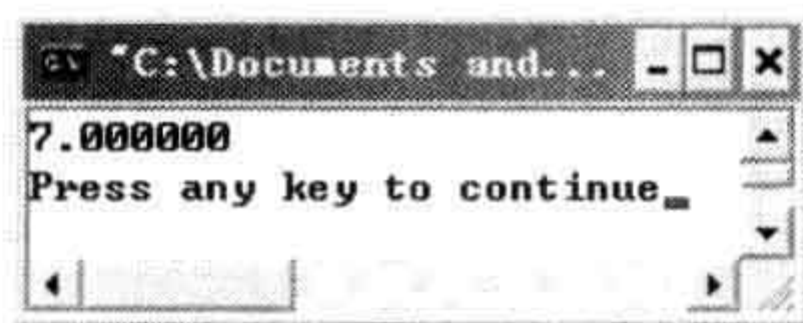


图 15.5 无参宏定义的运行结果

上例中,首先进行宏定义,定义 A 表达式和 P 输出格式,然后在主函数中做了宏调用。在预处理时,经宏展开后的语句变为 $z=x*x+2/y+3$,然后将主函数中定义的变量代入表达式,得到运算结果,最后宏调用 P,输出结果。

注意:

在宏定义时,语句结束不要加分号,因为分号在宏定义中并不表示宏定义的结束,而是会被当做字符替换掉。



Note



专家点评

无参宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，只是一个简单的代换，在字符串中可以包含任意字符，在预处理过程对它不做任何错误检查，只有在编译已经被宏展开后的源程序中才会发现错误。



Note

问题 265 什么是带参宏定义？

问题阐述

什么是带参宏定义？如何应用？

专家解答

带参宏定义，顾名思义，就是带有参数的宏定义。在宏定义中的参数称为形式参数，在宏调用中用的参数称为实际参数。对带参数的宏，在调用中，不仅是进行简单的字符串代换，而且还要进行参数替换。带参宏定义的一般形式为：

```
#define 宏名(参数表) 字符串
```

带参数的宏定义的展开置换过程为：

在程序中，如果有带实参的宏，则按#define 命令行中指定的字符串从左到右进行置换。如果串中包含宏中的形参，则将程序语句中相应的实参代替形参；如果宏定义中的字符串中的字符不是参数字符，则保留。这样就得到了置换的字符串。

通过下面的例子，了解带参宏定义的应用。相应代码如下。

```
#define MIN(a,b) (a<b)?a:b
main()
{
    int m,n,min;
    printf("输入两个数: ");
    scanf("%d,%d",&m,&n);
    min=MIN(m,n);
    printf("min=%d\n",min);
}
```

程序的运行结果如图 15.6 所示。

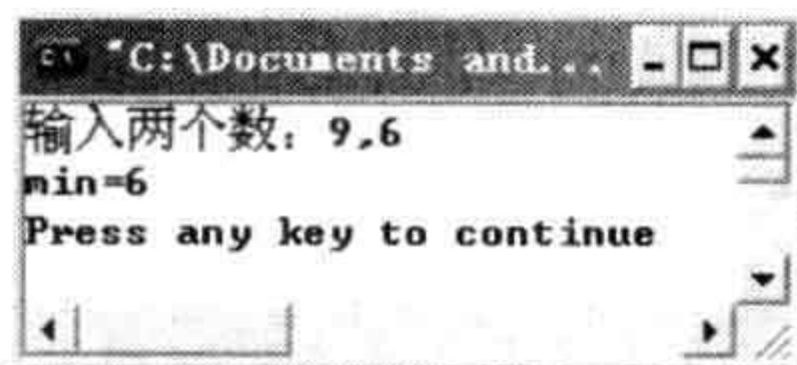


图 15.6 带参宏定义的应用



上例为在宏定义中定义了一个比较两个数大小的宏名,通过主函数中的 `min=MIN(m,n)` 进行宏调用,实参 `m`、`n` 代换形参 `a`、`b`。宏展开后该语句为: `min=(m<n)?m:n`;用于计算 `m` 和 `n` 中的较小数。

注意:

在带参宏定义中,宏名和形参表之间不能有空格出现。



Note

专家点评

由带参的宏可以联想到带参的函数,两者有些相似,但是本质上是有很区别的。通过本问题的学习,要掌握带参宏的应用。

问题 266 怎样写参数个数可变的宏?

问题阐述

在 C 语言中存在参数个数可变的函数,那么是否也存在参数个数可变的宏呢?如果存在,怎样写参数个数可变的宏呢?

专家解答

在 C 语言中存在参数个数可变的宏,首先大致了解一下什么是参数个数可变的函数, `printf()` 和 `scanf()` 函数是使用最频繁的参数个数可变的函数。

下面这个例子是参数个数可变的函数的应用,相应代码如下。

```
#include<stdio.h>
#include<stdarg.h>
int func(int first, int second, ...)
{
    int s = 0, t = first;
    va_list vl;
    va_start(vl, first);
    while (t != -1)
    {
        s+= t;
        t= va_arg(vl, int);          /*将当前参数转换为 int 类型*/
    }
    va_end(vl);
    return s;
}
int main(int argc, char* argv[])
{
    printf("个数不定的参数的和为 s= %d\n", func(50,10,12, 20,-1)); /*-1 是参数结束标志*/
}
```




```
printf("个数不定的参数的和为 s= %d\n", func(50,10,12, 20,8,-1)); /*-1 是参数结束标志*/
return 0;
}
```



Note

此例实现了一个参数个数不确定的求和的函数 `func()`，在 `func()` 函数中输入任意个数的参数，输出这几个参数的和。例如，输入参数 50、10、12、20、-1，可以得到这几个参数的和为 92。其中，-1 是参数结束标志。当输入 50、10、12、20、8、-1 时，函数得到的计算结果为 100。此程序的运行结果如图 15.7 所示。

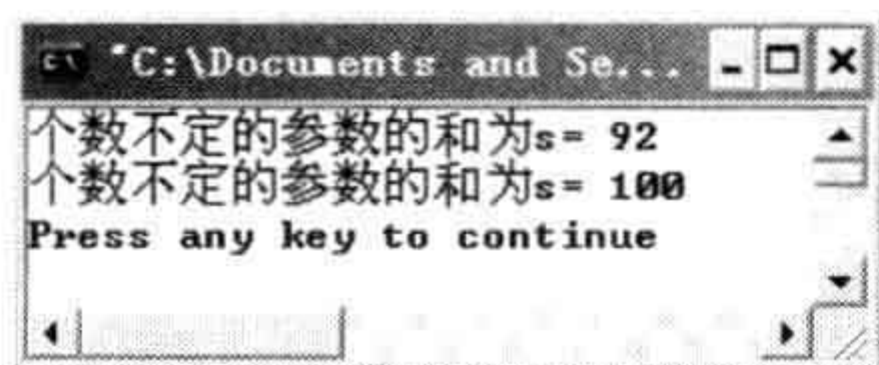


图 15.7 参数个数可变的函数

上述例子中，函数中一开始定义了一个 `va_list` 型变量 `vl`，该变量用来访问可变参数，实际上就是指针，接着使用 `va_start` 使 `vl` 指向第一个参数，然后再使用 `va_arg` 来遍历每一个参数，`va_arg` 返回参数列表中的当前参数并使 `vl` 指向参数列表中的下一个参数。最后通过 `va_end` 把 `vl` 指针清为 `NULL`。在这里，`va_start`，`va_arg`，`va_end` 其实都是宏。

参数个数可变的宏与参数个数可变的函数大体是相似的，都是参数个数不确定。了解了如何编写一个参数个数可变的函数，那么如何编写一个参数个数可变的宏也就可以理解了。

一种流行的技巧是用一个单独的用括号括起来的“参数”定义和调用宏，参数在宏扩展的时候成为类似 `printf()` 那样的函数的整个参数列表。例如：

```
#define DEBUG(args) (printf("DEBUG:"),printf args)
If(n!=0)
DEBUG(("n is %d\n",n));
```

注意：

调用者必须记住使用一对额外的括号括起，如 “`DEBUG(("n is %d\n",n));`”。

GCC 有一个扩展可以让函数式的宏接受可变个数的参数。但这并不是标准，另一种可能的解决方案是根据参数个数使用多个宏，如 `DEBUG1`、`DEBUG2` 等，或者用逗号表示成如下形式。

```
#define DEBUG(args) (printf("DEBUG:"),printf args)
#define _ ,
DEBUG("i=%d" _ i);
```

C99 引入了对参数个数可变的函数式宏的正式支持。在宏“原型”的末尾加上符号“...”，宏定义中的伪宏 `_VA_ARGS_` 就会在调用时替换成可变参数。例如：

```
#define dbgmsg(fmt,...)\
printf(fmt,_VA_ARGS_)
```




这里“...”表示可变参数。

专家点评

起初, 可变参数还只是应用在真正的函数中, 不能应用在宏中。直到 C99 编译器标准的出现, 改变了这种局面, 它允许定义参数可变的宏。虽然 GCC 也可以定义参数个数可变的宏, 但是它并不是标准的。



Note

问题 267 #pragma 预处理的作用是什么?

问题阐述

#pragma 预处理指令是所有预处理指令中最为复杂的, 有很多的参数, 不同的参数又有不同的作用, 下面了解一下 #pragma 预处理的各种作用。

专家解答

#pragma 预处理的作用是设定编译器的状态, 或者是指示编译器完成一些特定的动作。#pragma 指令对每个编译器给出了一个方法, 在保持与 C 和 C++ 语言完全兼容的情况下, 给出主机或操作系统专有的特征。依据定义, 编译指示是机器或操作系统专有的, 且对于每个编译器都是不同的。

该预处理命令的一般形式为:

```
#pragma para
```

其中 para 为参数, 不同的参数, 作用也会不同。下面介绍几种常见的参数。

(1) 参数为 message。

它的作用是能够在编译信息输出窗口中输出相应的信息, 这对于源代码信息的控制是非常重要的。其使用方法为:

```
#pragma message("消息文本")
```

这个参数还有一个功能, 就是在程序中如果忘记有没有正确地设置这些宏, 此时可以用这条指令在编译的时候进行检查, 输出相应的信息, 判断自己是否在源代码的什么地方定义了 _X86 这个宏。相应代码如下。

```
#ifdef _X86
#pragma message("_X86 macro activated!")
#endif
```

(2) 参数为 code_seg。

这个参数的作用是能够设置程序中函数代码存放的代码段, 在开发驱动程序的时候会使用它。



一般格式为：

```
#pragma code_seg([*section-name*[,*section-class*]])
```

(3) 参数为 `once`。

它的作用是只要在头文件的最开始加入这条指令，就能够保证头文件被编译一次。它的一般用法为：

```
#pragma once
```

(4) 参数为 `hdrstop`。

一般形式为：

```
#pragma hdrstop
```

它表示预编译头文件到此为止，后面的头文件不进行预编译。

(5) 参数为 `resource`。

一般表达形式为：

```
#pragma resource "*.dfm"
```

作用是把 `*.dfm` 文件中的资源加入工程。`*.dfm` 中包括窗体外观的定义。

常用到的参数有如上五种，除此之外，`#pragma` 预处理命令的参数还有 `warning`、`coment`、`pack` 等，在此不全部介绍。

专家点评

`#pragma` 预处理命令主要是通过各种参数，指示编译器完成特定的动作，要理解这些参数的作用并且熟悉其表达形式。

问题 268 条件编译的表达形式有哪些？

问题阐述

所谓的条件编译是指对源程序其中的一部分内容只在满足一定条件时才进行编译，也就是对一部分内容指定编译的条件。那么，条件编译的表达形式有哪些呢？

专家解答

1. `#ifdef` 标识符

程序段 1

`#else`

程序段 2

`#endif`

这第一种表达形式的功能是，如果标识符已被 `#define` 命令定义过，则对程序段 1 进行



编译；否则对程序段 2 进行编译。如果没有程序段 2，则 `#else` 语句可以省略。

2. `#ifndef` 标识符

程序段 1

`#else`

程序段 2

`#endif`

这第二种表达形式的功能是，若标识符未被定义过，则编译程序段 1，否则编译程序段 2。这种形式的作用与第一种形式正好相反。要记住 `ifndef` 是表示否定的含义，而 `ifdef` 是表示肯定的含义。若这种形式没有程序段 2，则同第一种一样可以省略 `#else` 语句。

3. `#if` 表达式

程序段 1

`#else`

程序段 2

`#endif`

它的作用是，若常量表达式的值为真（非 0），则对程序段 1 进行编译，否则对程序段 2 进行编译，因此可以使程序在不同条件下，完成不同的功能。

专家点评

条件编译的表达形式就有如上三种。不同的预处理命令有不同的功能，要根据在程序中的实际情况，对这三种表达形式进行选择。

问题 269 如何应用内部函数？

问题阐述

函数是 C 语言程序中的最小单位，往往把一个函数或多个函数保存到一个文件，这个文件就是源文件。已知函数是可以被调用的，但是当个源程序由多个源文件组成时，可以指定函数不能被其他文件调用，这种不能被其他函数所调用的函数称之为内部函数。那么，如何应用内部函数呢？

专家解答

在定义内部函数时，在函数名和函数类型的前面加上 `static`。表达形式如下：

`static` 类型标识符 函数名(形参表)

内部函数又被称为静态函数。该函数只被所在的源文件使用。如果在不同的文件中有同名的内部函数，则互不干扰。

使用内部函数的好处在于，不同的开发者可以分别编写不同的函数，不用担心所用函数是否会与其他文件中的函数同名。通常把只能由同一文件使用的函数和外部变量放在一



Note



个文件中，在它们面前都加上 static 关键字，使之局部化，导致其他文件不能引用。
通过一个内部函数的应用例子，熟悉内部函数的使用方法。相应代码如下。

```
#include<stdio.h>
static char* GetString(char* pString)           /*定义赋值函数*/
{
    return pString;                             /*返回字符*/
}
static void ShowString(char* pString)           /*定义输出函数*/
{
    printf("%s\n",pString);                     /*显示字符串*/
}
int main()
{
    char* pMyString;                            /*定义字符串变量*/
    pMyString=GetString("MingRi!");            /*调用函数为字符串赋值*/
    ShowString(pMyString);                     /*显示字符串*/
    return 0;
}
```

此例的功能为调用一个自定义的 GetString 函数获得字符串，然后通过调用 ShowString 函数输出获得的字符串。这两个自定义的函数均在前边加上了 static 做修饰，使这两个函数只能在其源文件中进行调用。

程序的运行结果如图 15.8 所示。

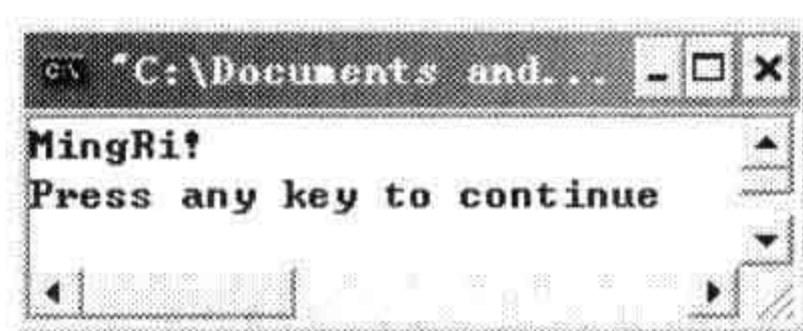


图 15.8 内部函数的应用

专家点评

在学习内部函数时，要了解内部函数的意义与使用方法，这样在以后的编程中才会将此知识应用自如。

问题 270 如何应用外部函数？

问题阐述

什么是外部函数？如何应用外部函数呢？

专家解答

所谓外部函数，就是放在外面的函数，放在共用位置的函数，这样谁有需要就可以调



用此函数，就像是公共电话。

外部函数就是可以被其他源文件调用的函数。定义一个外部函数的一般形式为：

extern 类型标识符 函数名(形参表)

在定义一个函数时，若没有加 **extern** 关键字，则默认为是外部函数。平时编写的很多自定义函数都是外部函数。

通常在需要调用此函数的文件中，用 **extern** 声明所用的函数是外部函数。

此例使用外部函数实现获取字符串，并且读取此字符串的功能。获取字符串函数和读取字符串函数分别定义在不同的文件中。代码如下。

```

/*////////////////////////////////////*/
/*                                file.c                                */
/*////////////////////////////////////*/
#include<stdio.h>
#include"externfile1.c"
#include"externfile2.c"
extern char* GetString(char* pString);          /*声明外部函数*/
extern void ShowString(char* pString);          /*声明外部函数*/
int main()
{
    char* pMyString;                            /*定义字符串变量*/
    pMyString=GetString("MingRi!");             /*调用函数为字符串赋值*/
    ShowString(pMyString);                       /*显示字符串*/
    return 0;
}
/*////////////////////////////////////*/
/*                                externfile1.c                        */
/*////////////////////////////////////*/
extern char* GetString(char* pString)
{
    return pString;                             /*返回字符*/
}
/*////////////////////////////////////*/
/*                                exterbfile2.c                        */
/*////////////////////////////////////*/
extern void ShowString(char* pString)
{
    printf("%s\n",pString);                     /*显示字符串*/
}

```

程序的运行结果如图 15.9 所示。

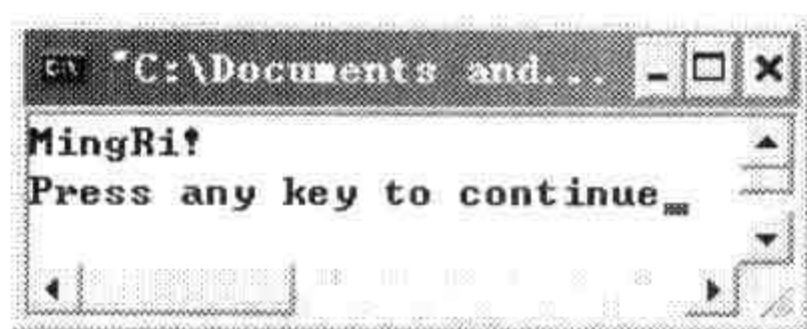


图 15.9 外部函数的应用





此例中，主函数存放在源文件 `file.c` 中，实现对功能函数的调用；获得字符串功能的外部函数存放在 `externfile1.c` 中；实现显示字符串功能的外部函数存放在 `externfile2.c` 中。

专家点评

外部函数应用广泛，要熟练掌握外部函数的使用方法。学习外部函数同时也要了解如何运行一个多文件程序。运行一个多文件程序有很多方法，在此例中采用了在主函数的开头使用 `#include` 命令包含了外部函数所在的文件。



Note

第16章

文件的读写操作

- ▶▶ 各个读写操作的区别是什么？
- ▶▶ C语言文件有哪几类？
- ▶▶ 怎样写数据文件，使之可以在不同字大小、字节顺序或浮点格式的机器上读入？
- ▶▶ 能否使用 `fflush()` 函数清除多余的输入？
- ▶▶ `fopen()` 函数打开文件失败的原因是什么？
- ▶▶ 为什么打开文件后要及时关闭？
- ▶▶ 文件的打开方式有哪些？
- ▶▶ 如何正确使用 `putchar()` 函数和 `getchar()` 函数？
- ▶▶ `getchar()` 函数、`getch()` 函数和 `getc()` 函数的区别是什么？
- ▶▶ 使用 `printf()` 函数和 `scanf()` 函数需要注意什么？
- ▶▶ `printf()` 函数有哪些参数？
- ▶▶ `scanf()` 函数的格式控制包括哪些？
- ▶▶ `printf()` 函数和 `scanf()` 函数格式符的修饰符 “*” 有什么作用？
- ▶▶ `fscanf()` 函数、`fprintf()` 函数与 `scanf()` 函数和 `printf()` 函数有什么不同？
- ▶▶ 如何判断文件的结束？



问题 271 各个读写操作的区别是什么

问题阐述

在 C 语言中, 打开一个文件, 对这个文件的操作包括: 读出数据、写入数据、文件定位和出错检测几种。主要的操作是对文件的读写操作, 读写操作有很多种, 那么多种读写操作存在什么区别呢?

专家解答

文件的读写操作有 `fputc()` 函数和 `fgetc()` 函数、`fread()` 函数和 `fwrite()` 函数、`fprintf()` 函数和 `fscanf()` 函数, 还有很多其他的读写函数, 下面对这几种读写操作分别介绍:

1. 字符读写函数 `fgetc()` 和 `fputc()`

字符读写函数是以字符为单位的读写函数。每次可从文件读出或向文件写入一个字符。

(1) `fgetc()` 函数的作用是从指定的文件中读一个字符, 函数调用的形式为:

```
c=fgetc(fp);
```

表示从打开的文件 `fp` 中读取一个字符并赋给 `c`。

对于 `fgetc()` 函数的使用需要注意以下几点:

- ☑ 在 `fgetc()` 函数调用中, 读取的文件必须是以读或读写方式打开的。
- ☑ 读取字符的结果可以不向字符变量赋值, 但是读出的字符不能保存。

(2) `fputc()` 函数的作用是把一个字符写入指定的文件中, 函数调用的形式为:

```
fputc(c,fp);
```

其中, `c` 为要输出的字符, 可以是一个字符常量, 也可以是字符变量; `fp` 是文件指针变量。

这个函数表示将字符 (`c` 的值) 输出到 `fp` 所指的文件中。此函数的返回值就是输出的字符, 如果输出失败, 则返回一个 `EOF(-1)`, `EOF` 是在 `stdio.h` 头文件中宏定义的常量, 值为 `-1`。

使用 `fputc()` 函数需要注意如下两点:

- ☑ 被写入的文件可以用写、读写、追加方式打开。
- ☑ 每写入一个字符, 文件内部位置指针向后移动一个字节。

2. 读写数据块的函数 `fread()` 和 `fwrite()`

读写数据块的函数 `fread()` 和 `fwrite()` 的一般调用形式为:

```
fread(buffer,size,count,fp);  
fwrite(buffer,size,count,fp);
```




其中, 参数 `buffer` 表示一个指针, 是读入数据的存放地址或者输出数据的地址; 参数 `size` 表示要读写的字节数; 参数 `count` 表示要进行读写多少个 `size` 字节的数据; 参数 `fp` 是文件型指针。

例如:

```
fread(p,4,2,fp);
```

此例表示从 `fp` 所指的文件中, 每次读 4 个字节 (一个实数) 送入实数组 `fa` 中, 连续读 2 次, 即读 2 个实数到 `fa` 中。

使用读写数据块的函数 `fread()` 和 `fwrite()` 时需要注意, 它们一般用于二进制文件的输入与输出, 因为它们是按照数据块的长度来处理输入与输出的, 在字符发生转换时容易出现与设想不同的情况。

3. 格式化读写函数 `fscanf()` 和 `fprintf()`

这两个函数与输入输出函数 `scanf()` 和 `printf()` 很相似, 但是两者的区别是 `fscanf()` 函数和 `fprintf()` 函数是用于读写磁盘文件中的数据, 而不是读写终端数据。它们的一般调用形式为:

```
fscanf(文件指针, 格式字符串, 输入列表);  
fprintf(文件指针, 格式字符串, 输出列表);
```

例如:

```
fscanf(fp,"%d,%f",&a,&b);
```

如果磁盘文件上存有 10, 6.81, 则此函数表示将 `fp` 所指的磁盘文件中的整数 10 送给变量 `a`, 将实数 6.81 送给变量 `b`。

```
int a=6;float b=4.56;  
fprintf(fp,"%d,%5.3f",a,b);
```

上例表示向 `fp` 所指的磁盘文件中写入一个整型数字 6 和一个浮点型数字 4.56。在磁盘上会显示为:

```
6, 4.560
```

在使用此函数时, 由于在输入时要把 ASCII 码转换为二进制, 在输出时又要把二进制形式转换为字符, 花费在转换上的时间比较多。因此, 在内存与磁盘频繁交换数据的情况下, 尽量不要使用此函数。

专家点评

在 C 语言中还存在很多其他的读写函数, 例如, 读写字符串的函数 `fgets()` 和 `fputs()` 等, 因为不常用到所以不做过多介绍。在上述介绍的读写函数中, 要注意区分它们的用法, 以及打开文件的形式。



Note



问题 272 C 语言文件有哪几类?

问题阐述

所谓文件,是指一组相关数据的有序集合。对文件的这个定义比较笼统,从不同的角度理解文件,就会对文件有不同的分类,那么 C 语言的文件都从哪几种不同的角度进行理解,分成了哪几类呢?

专家解答

在 C 语言中,经常提到的文件有源文件、目标文件、可执行文件、磁盘文件、文本文件、二进制文件等。那么,这些文件都是同一类型的吗?依据什么将文件分成了这么多的类型呢?下面具体介绍。

(1) 从文件的内容角度理解,可以将文件分为源文件、目标文件和数据文件。源文件中存放的是 C 语言的源代码;目标文件存放的是将源代码编译之后的结果;数据文件是指先将数据写入一个文件中,需要时从中读取数据的文件。

(2) 按文件的组织形式来分,可以将文件分为顺序存取文件和随机存取文件。所谓的顺序存取文件是指从文件的开头对数据进行逐个地读写操作;而随机存取文件是指从任意指定位置进行数据的读写操作。

(3) 根据对文件的处理方法的不同,可将文件分为缓冲区文件和非缓冲区文件。所谓缓冲区文件是指系统自动地在内存区为每一个正在使用的文件开辟一个缓冲区。而非缓冲区文件是指不自动地开辟确定大小的缓冲区,而由程序本身为每个文件设定缓冲区。

从内存向磁盘输出数据时,必须先将数据传送到内存中的缓冲区,待装满缓冲区后,再将数据一起送到磁盘。

(4) 根据文件的操作方法的不同可将文件分成标准文件和一般文件。标准文件是指系统自动为缓冲区分配的文件,这种文件只有三种类型,分别是标准输入文件、标准输出文件和标准出错信息文件。

一般文件是指除了标准文件之外的磁盘文件和设备文件。在使用这类文件时,需要先用文件打开函数将文件打开;在不使用此文件时,需要及时关闭,否则会造成数据的不安全。

(5) 根据数据的组织形式的不同,可将文件分为文本文件和二进制文件。在 C 语言中,把文件看作是一个字符序列,即由一个个字符的数据顺序组成的文本文件和二进制文件。

文本文件又称为 ASCII 文件,它的每一个字节存放一个 ASCII 代码,代表一个字符,二者是一一对应的。因而,此文件便于对字符进行逐个处理,也便于输出字符,但是占用的存储空间比较多,而且要花费 ASCII 码与二进制形式间的转换时间。

二进制文件是把内存中的数据按其所在内存中的存储形式,原样输出到磁盘上存放,占用字节比较少,并且不需要转换,但是一个字节并不对应一个字符,不能直接输出字符



形式。

专家点评

通过上面对文件的分类理解,可以帮助初学 C 语言的程序员了解常用文件的含义,例如,源文件、二进制文件、文本文件等。



Note

问题 273 怎样写数据文件,使之可以在不同字大小、字节顺序或浮点格式的机器上读入?

问题阐述

怎样写数据文件,使之可以在不同字大小、字节顺序或浮点格式的机器上读入,也就是说怎样写一个可移植性好的数据文件?

专家解答

最好的移植方法是使用文本文件,它的每一字节放一个 ASCII 代码,代表一个字符。

用文本文件的形式输出与字符一一对应,一个字节代表一个字符,便于对字符进行逐个处理,也便于输出字符。

例如,存放一个整数 25697 在内存中以文本文件形式存储的效果如图 16.1 所示。

00110010	00110101	00110110	00111001	00110111
----------	----------	----------	----------	----------

图 16.1 文本形式存储整数 25697

以二进制文件的形式存储的效果如图 16.2 所示。

01100100	01100001
----------	----------

图 16.2 二进制文件形式存储整数 25697

可见,以文本形式存储这个整数需要占用五个字节。而以二进制文件形式存储仅需两个字节。虽然文本文件形式存储占用的内存比二进制文件占用的内存大,但是可移植性比二进制文件高,在移植的时候可使用 `fprintf()` 函数输出, `fscanf()` 函数读入。

专家点评

很多人认为文本文件太大,使用起来读写太慢,但是通常计算机操作的效率是可以接



受的。

问题 274 能否使用 fflush() 函数清除多余的输入?

问题阐述

在从终端输入数据时,很可能会输入多余的数据,那么能否使用 fflush() 函数清除呢?

专家解答

fflush() 函数只是用在文件以写的方式打开时,将缓冲区内容写入到文件。因此 fflush() 函数仅对输出流有效,对输入流并不能用于放弃剩余的输入。可通过如下代码了解 fflush() 函数是否清除了缓冲区中多余的输入。

```
int integer;
char string[81];
printf( "Enter a sentence of four words with scanf: " );
for( integer = 0; integer < 4; integer++ )
{
    scanf( "%s", string );
    printf( "%s\n", string );
}
fflush( stdin );                                /*丢弃输入缓冲区中的内容*/
printf( "Enter the same sentence with gets: " );
gets( string );
printf( "%s\n", string );
```

此例的运行结果为:

```
Enter a sentence of four words with scanf:i am a girl !!!!
i
am
a
girl
Enter the same sentence with gets:!!!!
```

由此可知, fflush() 函数在清除缓冲区中的内容时,并没有将缓冲区中多余的数据清除,使得调用 gets() 函数获取字符串时获取了多余的数据 “!!!!”。

说明:

fflush() 函数非标准库函数,因此在部分编译环境中可以清除多余的字符,在部分环境中不可以清除多余的字符,例如,在 VC6.0 编译环境中可以清除多余的输入字符,但是使用 gcc 编译程序时, fflush() 函数不可以清除多余的输入字符。上述运行结果是在 gcc 环境中编译得出的结果。



在 gcc 编译环境中, `fflush()` 函数无法清除多余的输入, 那么此时要怎样清除多余的输入呢?

若是希望清除掉 `scanf()` 函数之后所剩下的换行符和其它的未知输入, 需要重写 `scanf()` 函数, 或者使用代码语句, 屏蔽掉后面的字符, 例如:

```
while((ch=getchar())!='\n'&&ch!=EOF)
/*清除*/;
```

没有什么标准的方法能够清除掉标准输入流中未读取的字符, 因为未读取字符也可能来自其他的操作系统的输入缓冲区。

专家点评

若需要严格丢弃多余输入的字符, 可以参见使用的系统的相关技术。

问题 275 `fopen()` 函数打开文件失败的原因是什么?

问题阐述

应用 `fopen()` 函数打开文件, 有的时候即使给出了详细的路径, 也会出现错误, 这是什么原因呢?

专家解答

文件打不开可能有两种情况:

(1) 指定位置不存在此文件。但是如果以带有字符 `w` 的方式打开文件时, 就不会出现打不开文件的情况, 因为字符 `w` 代表的是向文件中写数据, 若存在路径指定的文件则删除文件重新创建一个新的文件, 向里面写入数据; 如果指定路径下不存在此文件, 则会自动创建一个新文件。但是, 其他打开文件的方式就不是了, 必须在路径下存在文件, 才能成功打开文件。

(2) 文件打不开还有一种情况, 就是路径的输入上有错误, 如这个打开函数中输入的路径:

```
fopen("c:\newfile\file.txt","r")
```

在此文件夹下确实存在这个文件 `file.txt`, 但是始终打开失败, 原因可能是文件名中的反斜杠与紧跟在后面的字符形成特殊的含义, 如 “`\n`”。为了避免这种情况, 在输入路径的时候, 可以输入双反斜杠。例如:

```
fopen("c:\\newfile\\file.txt","r")
```

避免了上述两种情况后, 就可以顺利打开文件了。



Note



专家点评

在打开文件时，要根据后面用到的文件读写函数的要求，来选择打开文件的方式。



Note

问题 276 为什么打开文件后要及时关闭？

问题阐述

对文件读写之前，需要使用 `fopen()` 函数打开该文件；在使用完文件后，需要及时关闭文件。那么，为什么要及时关闭文件呢？

专家解答

文件打开的过程是将存放在磁盘等外部设备上的文件读入到内存中，以方便访问，对其进行操作。有了文件的打开就要有文件的关闭，否则文件会被误用。

若文件做了一定的修改或操作后不再使用，则需要及时将文件关闭，否则会丢失文件的数据，改动也会丢失。因为向文件写数据时，是先将数据输入缓冲区，等缓冲区满后，才可以输出给文件，如果当数据未充满缓冲区，而程序却结束了运行，就会将缓冲区中的数据丢失。而如果及时地将文件关闭，则会避免这个问题。因为文件的关闭过程除了关闭所有的文件以外，还会先把缓冲区中的数据输出到磁盘文件，然后才释放文件指针变量。

及时地关闭文件不仅会保证打开文件不会丢失数据，还会增加机器的运行效率。因为在打开文件时会占用一定的内存空间，如果文件打开以后不及时关闭，占用内存会越来越多，可以使用的内存就会越来越少，从而影响机器的运行效率。

专家点评

做任何事情都要学会始有终。既然打开了文件，在不使用此文件时，就要及时地关闭文件。

问题 277 文件的打开方式有哪些？

问题阐述

在程序中运行文件相关的读写操作时，需要首先将文件打开，不同的读写操作，对文件的打开方式有着不同的要求，那么文件的打开方式都有哪些呢？

专家解答

文件的打开方式如表 16.1 所示。



表 16.1 文件的打开方式

文件的打开方式	作 用
r	以只读的方式打开一个文本文件，只允许读数据
w	以只写的方式打开或创建一个文本文件，只允许写数据
a	以追加的方式打开一个文本文件，并在文件末尾写数据
rb	以只读的方式打开一个二进制文件，只允许读数据
wb	以只写的方式打开或创建一个二进制文件，只允许写数据
ab	以追加的方式打开一个二进制文件，并在文件末尾写数据
r+	以读写的方式打开一个文本文件，允许读和写
w+	以读写的方式打开或创建一个文本文件，允许读和写
a+	读写打开一个文本文件，允许读或者在文件末尾追加数据
rb+	以读写的方式打开一个二进制文件，允许读和写
wb+	以读写的方式打开或者创建一个二进制文件，允许读和写
ab+	读写打开一个二进制文件，允许读或者在文件末尾追加数据



Note

上述为十二种文件的打开方式，观察这十二种打开方式，其中字符 r 表示读（read），字符 w 表示写（write），字符 a 表示追加（append），字符 b 表示二进制文件（binary），字符“+”表示读和写操作。因此根据几种字符的组合，规定出了上表中的十二种打开文件的方式。

使用文件打开方式时需要掌握如下几点：

（1）在使用带有字符 w 的打开方式时，若指定的文件原来不存在，则可以在打开时，先创建一个文件；若文件存在，则在打开时要将该文件删除，然后重新创建一个新文件。

（2）在使用带有字符 r 的打开文件方式时，若指定的文件原来不存在，在运行程序时会出现错误；若文件存在，则与带有字符 w 的打开文件方式相同，先删除原有文件，再创建新的文件。

（3）在使用带有字符 a 的打开方式时，若指定的文件不存在，程序运行会出错；若文件存在，则打开文件，并在文件的末尾追加数据，不可以删除文件再创建新的文件。

（4）不带字符 b 的打开方式表示打开的是文本文件，带字符 b 的表示打开的是二进制文件。

（5）带有字符“+”的表示既允许读数据，也允许写数据。

专家点评

要记住各个字符代表的含义，这样在记忆这十二种文件的打开方式时会更容易记住，且不容易记混。在选择文件的打开方式时，要符合程序中用到的文件函数的要求。



问题 278 如何正确使用 putchar() 函数和 getchar() 函数?

问题阐述

putchar() 函数和 getchar() 函数是两个标准的输出、输入函数。那么这两个函数如何正确使用呢?

专家解答

getchar() 函数和 putchar() 是标准 I/O 函数库中最容易理解的字符输入与输出函数。putchar() 函数用于向终端输出一个字符, 一般格式为:

```
putchar(ch);
```

它输出字符变量 ch 的值, ch 可以是字符型变量, 也可以是整型变量, 并且还可以输出控制字符, 如回车符, 但是整型变量也是以相对应的字符的形式输出, 控制字符则起到相应的作用。例如:

```
#include<stdio.h>
main()
{
    char a,b,c;
    int i;
    a='h';
    b='o';
    c='w';
    i=48;
    putchar(a);
    putchar(b);
    putchar(c);
    putchar('\n');
    putchar(i);
    putchar('\n');
}
```

此程序输出了三个单个字符, 输出了两个回车符, 输出了一个整型数, 但是整型数根据 ASCII 码表将其转换为相对应的字符形式, 程序运行结果如图 16.3 所示。

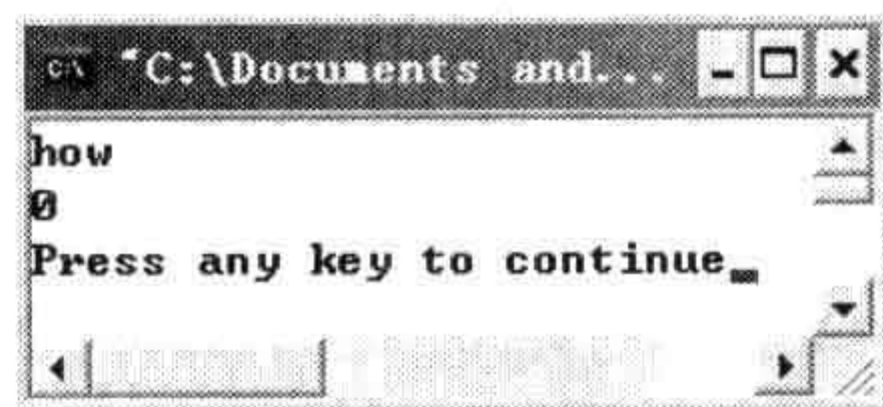


图 16.3 putchar 函数的应用



`getchar` 函数用于从终端输入一个字符，该函数中没有参数。该函数的返回值就是从键盘输入的单个字符。使用 `getchar()` 函数需要注意，此函数只能读取一个字符，如果在键盘上输入多个字符，则此函数只取第一个字符，并且这个获取单个字符的函数在 Turbo C 系统中运行带有此函数的程序时，会退出 Turbo C 屏幕，进入用户屏幕，等待用户输入完字符，则返回到 Turbo C 屏幕。

专家点评

在 C 标准 I/O 函数库中的两组输入输出函数 `getchar()` 函数、`putchar()` 函数和 `scanf()` 函数、`printf()` 函数，其中前两个字符输入输出函数在程序中使用时需要包含所在的头文件 `stdio.h`，否则会出错，而 `printf()` 函数和 `scanf()` 函数则可以不用包含头文件，不会出错，只会提示警告信息，但不影响程序的运行。

问题 279 `getchar()` 函数、`getch()` 函数和 `getche()` 函数的区别是什么？

问题阐述

很多程序员的代码有的时候用到 `getchar()` 函数，有的时候用到 `getch()` 函数，很多初学者认为这两个函数的作用都是读取一个从键盘输入的字符，两者有什么区别呢？

专家解答

`getchar()` 函数的作用是从终端输入一个字符，返回从输入设备得到的字符。`getch()` 函数的作用也是从终端输入一个字符，但是两者的区别在于 `getchar()` 函数具有缓冲区，并且在使用此函数时，会将从键盘输入的数据显示在屏幕上，这可以称之为回显（就像是手机的来电显示）。而 `getch()` 函数既没有缓冲区，也没有回显的功能。

所谓的缓冲区其实就是在内存中开辟一个临时单元，用来存放键盘输入的字符，输入结束时，`getchar()` 函数将从缓冲区读取这个字符。

在程序代码中，有时还会看到 `getche()` 函数，似乎与这两个函数的功能相似。是的，这个函数与 `getch()` 函数都是在 Turbo C 系统中增加的两个读取单个字符的函数，`getche()` 函数与上述两个函数也各不相同，它没有缓冲区，但是带有回显的功能。

了解了上述三种读取单个字符的函数的区别，可以在编写程序或者理解代码的时候知道为什么要选用这个函数了。例如，在用 Turbo C 系统中编写游戏时，需要输入任意键结束游戏，则可以使用 `getch()` 函数，这样不会在屏幕上显示输入的任意字符，显示了也没有意义，只会使界面不美观；然而在需要起到提示作用的时候，就需要使用 `getchar()` 或者 `getche()` 函数，提示用户，已经正确输入了一个字符。



Note



专家点评

这三个功能基本相同的函数，可以根据程序员自己的需要选择使用，这为 C 语言编程带来了方便与灵活。



Note

问题 280 使用 printf() 函数和 scanf() 函数需要注意什么？

问题阐述

使用格式输入输出函数都需要注意什么，即使用 printf() 函数和 scanf() 函数应该注意什么？

专家解答

格式输入输出函数的原型在头文件 `stdio.h` 中，但是通常在程序中使用这两个格式输入输出函数可以不包含 `stdio.h` 头文件。在 C 语言中，每一个函数的使用都有其相应的规定及注意事项，下面了解一下经常用到的 `printf()` 函数和 `scanf()` 函数都有哪些需要注意的。

1. 使用 `printf()` 函数需要注意以下几点。

(1) `printf()` 函数的格式控制字符串由格式字符串和非格式字符串组成，其中格式字符串是由 %、修饰符和各种数据类型的格式字符组成。在使用时要注意修饰符的意义，以及格式字符代表的数据类型。同时，在输出函数中的非格式字符串要原样输出，用以起到提示说明的作用。

(2) `printf()` 函数的参数列表代表要输出的参数，要求要输出的参数与格式字符一一对应，并且参数列表代表的是要输出的数据，可以是表达式。

2. 使用 `scanf()` 函数需要注意以下几点。

(1) 此输入函数中双引号后面的内容应当是需要输入的变量的地址，而不是变量名，如：

```
scanf("%d%d",&a,&b);
```

`&a` 和 `&b` 代表整型变量 `a` 和 `b` 的地址。

(2) 如果在格式控制字符串中除了格式字符还有其他的非格式字符，则在键盘输入时，也要原样输入，例如：

```
scanf("a=%d,b=%d",&a,&b);
```

运行程序时，通过键盘向终端输入时，要按照如下形式：

```
a=3,b=5
```

若不按照格式控制中的字符串形式输出，则会运行出错。

(3) `scanf()` 函数不同于 `printf()` 函数，`scanf()` 函数在格式控制字符串中不可以设定精度，若写成如下形式是错误的：



```
scanf("%3.2f",&a);
```

(4) 在输入字符时, 使用“%c”的格式字符, 此时的空格字符和转义字符在输入时都作为有效的单个字符表示, 例如:

```
scanf("%c%c",&a,&b);
```

若在键盘输入时用空格将两个字符隔开, 则在读取键盘的输入时会将空格作为第二个要读取的字符, 即若输入形式如下:

D 空格 c

则字符“D”赋给变量 a, 字符“空格”赋给变量 b。所以要注意在要求读入字符时, 不需要使用空格作为两个字符的间隔。

(5) 要注意在输入数据时, 如果遇到下述情况, 则输入结束。如:

- ☒ 若输入空格、回车符或跳格符 (Tab 键) 时, 就结束数据的输入;
- ☒ 若指定了输入的宽度, 则到达规定的宽度, 自动结束数据的输入;
- ☒ 遇到非法输入时, 结束数据的输入。

专家点评

这两个格式输入输出函数的应用非常广泛, 很多初学者最会用的就是这两个输入输出函数, 然而也可以说是最不会用这两个输入输出函数, 因为在使用时往往忽略了上面应该注意的事项。

问题 281 printf()函数有哪些参数?

问题阐述

printf()函数的作用是向终端输出若干个任意类型的数据, 此函数由格式控制部分和输出表列两部分组成, 格式控制部分又由“%”和格式字符串组成, 那么, 此函数格式字符串部分有哪些参数呢?

专家解答

printf()函数的格式字符串由修饰符和类型组成。

printf()函数的类型表示输出的数据类型。类型的格式符和意义如表 16.2 所示。

表 16.2 printf()函数的格式符和意义

格 式 符	意 义
d	以十进制形式输出有符号整数 (正整数不输出符号)
u	以十进制形式输出无符号整数
o	以八进制无符号形式输出整数 (不输出前导符0)
x	以十六进制形式输出无符号整数 (不输出前缀0x)



Note



续表

格 式 符	意 义
f	以小数形式输出单、双精度实数
g	以指数形式输出单、双精度实数
c	输出单个字符
s	输出字符串
g	以%f或%e中较短的输出宽度输出单、双精度实数



Note

printf()函数的参数中还有好多起修饰作用的符号, 现在介绍几种常见的, 例如:

(1) 标志符号。标志字符有+、-、#、空格, 其中“-”号表示输出结果左对齐, 右边填充空格; “+”表示输出符号, 即正号或负号; “#”的应用对 c、s、d、u 这几个格式符没有影响, 但是对 o 格式符的影响是在输出时加前缀 0, 对 x 格式符, 在输出时加前缀 0x, 对 e、g、f 三个格式符, 当结果有小数时才给出小数点。

(2) 长度符号。长度字符有 h 和 l, h 表示按短整型量输出, l 表示按长整型量输出。

(3) 输出最小宽度。输出最小宽度并没有什么特定的字符, 而是用十进制整数来表示。用十进制整数表示输出的最少位数, 若实际位数多于定义的宽度, 则按实际位数输出; 若实际位数少于定义的宽度, 则补以空格或 0。

(4) 精度符号。精度格式符以“.”开头, 后跟十进制数。如果输出的是数, 表示小数点后保留几位小数的形式; 如果输出的是字符, 表示输出字符的个数; 若实际位数大于所定义的精度数, 则去掉超过的部分。

专家点评

向终端输出数据信息的函数 printf()是在程序中经常用到的, 关于它的参数大家一定要牢牢掌握, 这样才能写出更完美的代码。

问题 282 scanf()函数的格式控制包括哪些?

问题阐述

一个 scanf()函数由格式控制和地址表列两部分组成, 那么关于 scanf()函数的格式控制都包括什么?

专家解答

scanf()函数的格式控制的一般格式为:

%[修饰符]类型

类型指的是格式字符, 不同的数据类型用不同的格式字符, 表示常见类型的种类如表 16.3 所示。



表 16.3 scanf()格式说明中的类型

格 式 类 型	说 明
d	输入十进制整数
u	输入无符号十进制整数
o	输入无符号的八进制整数
x	输入无符号的十六进制整数
f或e	输入实型数
c	输入单个字符
s	输入字符串



Note

scanf()函数的格式说明字符串中的“[]”中的修饰符也有很多种,常见的如表 16.4 所示。

表 16.4 scanf()格式说明中的修饰符

字 符	说 明
l	用于输入长整型数据
h	用于输入短整型数据
域宽	用十进制整数指定输入的宽度,即字符数
*	表示在输入项读入后不赋予相应的变量,即跳过该项输入值

在了解了格式字符中的修饰符和类型两部分后,通过一个例子,对 scanf()函数的格式说明进一步理解,相应代码如下。

```
main()
{
    int a,b;
    scanf("%3d%5d",&a,&b);
    printf("a=%d\n",a);
    printf("b=%d\n",b);
}
```

运行程序,输入 12345678,显示结果如图 16.4 所示。

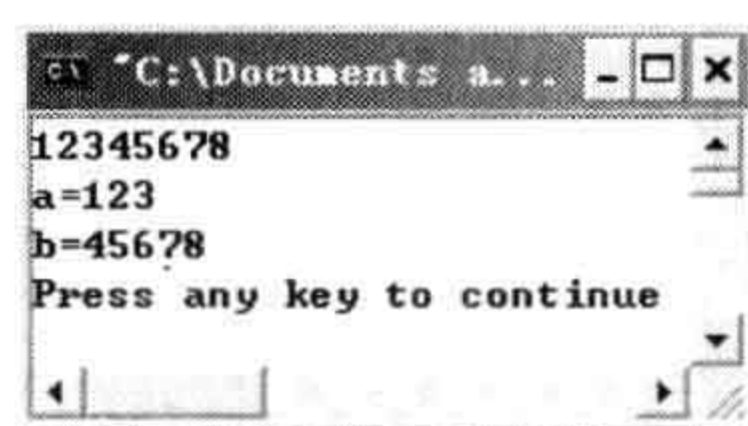


图 16.4 scanf()函数格式说明的应用

由于已经设定了域宽,因此无间断地输入 12345678 后,系统会自动将 123 赋给 a,将剩下的五位赋给 b。

专家点评

上述例子只介绍了域宽的应用,其余修饰符和格式字符的应用与此例相同,只是不同



的修饰符有不同的作用，不同的格式字符代表输入不同的数据类型数据而已。

问题 283 printf()函数和 scanf()函数格式符的修饰符“*”有什么作用？



Note

问题阐述

在 printf()函数和 scanf()函数的格式修饰符有很多，以浮点型数据为例，有%f、%lf、%3.0f、%.4f等。不同的修饰符表示不同的含义，那么修饰符“*”有什么含义呢？

专家解答

下面通过例子来证明一下这个格式符在 printf()函数中的作用是什么。相应代码如下。

```
main()
{
    int a,b;
    a=66,b=2;
    printf("%*d\n",b+2,a);
}
```

程序的运行结果如图 16.5 所示。

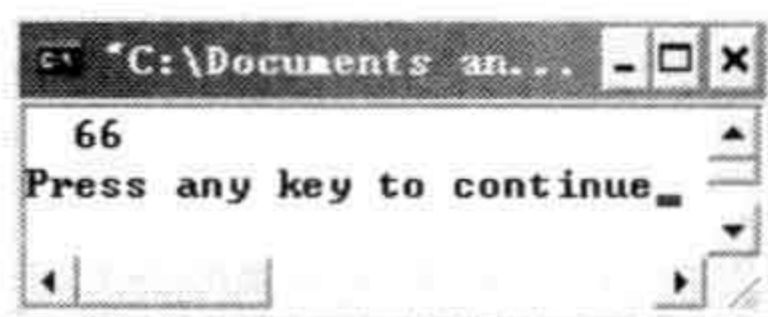


图 16.5 修饰符在 printf()函数中的作用

由此结果可以发现，整数 66 前面多出两个空格，这就是修饰符“*”在 printf()函数中起到的作用，即为指定的表达式的值给定输出项的域宽。本例中，输出项中给出的域宽为 b+2，也就是 4，因此去掉 66 占掉的两个宽度，前面还空下两个宽度。在此例中，相当于输出 %4d。

那么在 scanf()函数中的修饰符“*”与在 printf()函数中的作用相同吗？下面通过一个例子，来验证一下在两个函数中的作用是否相同，相应代码如下。

```
main()
{
    char a[80];
    scanf("%*2s%s",a);
    printf("a=%s",a);
}
```

程序的运行结果如图 16.6 所示。

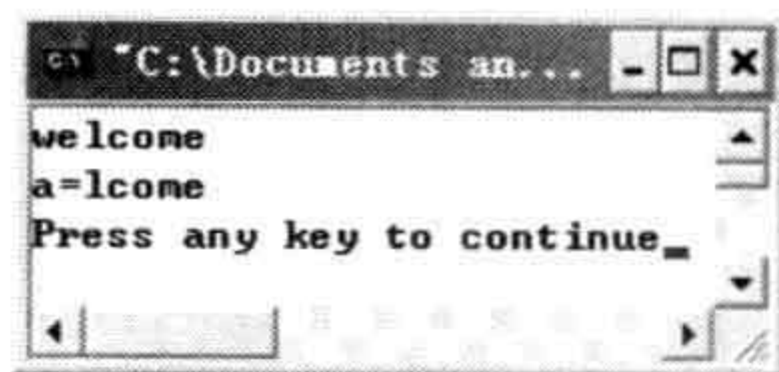


图 16.6 修饰符在 scanf()函数中的作用

由此结果可以发现，在键盘上输入“welcome”字符串，输出的结果却少了前面的两个字符“we”，这就是此修饰符在 scanf()函数中的作用，即“*2”表示跳过了输入字符的前两个字符。输入整数几就会跳过几个字符，若改为“*4”则会跳过前面四个字符，只输出显示 a=ome。

专家点评

关于这个格式修饰符，在终端文件的读写上很少用到，更多的时候会用到文件的读写操作中。

问题 284 fscanf()函数、fprintf()函数与 scanf()函数和 printf()函数有什么不同？

问题阐述

fscanf()函数、fprintf()函数与 printf()函数、scanf()函数的作用相似，都是格式化读写函数，那么这两个读写函数有什么不同呢？

专家解答

两者的区别就在于前面的字符“f”，即 fscanf()函数和 fprintf()函数的读写对象是磁盘文件（file），而不是键盘和显示器。

scanf()函数是通过键盘输入数据，使用 scanf()函数读取键盘上的输入信息；而 printf()函数是将信息输出到终端设备，即显示器上。

fscanf()函数是读取指定磁盘文件中数据信息；而 fprintf()是向指定的磁盘文件中输出信息，显示在磁盘文件上。

通过下面的例子，理解这四个函数的功能。代码如下。

```
#include<stdio.h>
int main()
{
    FILE *fp;
    long ln;
    float fl;
    char str[128];
    char str1[128];
    fp= fopen("time.txt","w+");           /*打开文件*/
```



Note



Note

```

fprintf(fp,"%s %ld %f","Hello",1100,12.34);      /*写入数据*/
fseek(fp,0L,SEEK_SET);                          /*设置文件指示器位置*/
fscanf(fp,"%s",str);                             /*读取字符串*/
fscanf(fp,"%ld",&ln);                           /*读取长整型数据*/
fscanf(fp,"%f",&fl);                             /*读取浮点型数据*/
printf("%s\t%ld\t%f\n",str,ln,fl);              /*输出数据*/
fclose(fp);
scanf("%s",str1);
printf("%s\n",str1);
}

```

此函数的功能是创建并打开一个磁盘文件 time.txt。通过 fprintf() 函数向此磁盘文件中写入数据,有字符串形式的 hello,有长整型形式的 1100,有浮点型的 12.34。然后使用 fseek() 函数设置文件指示器的位置,通过 fscanf() 函数读取此字符串,分别读取这几种类型的数据,然后将这些数据输出到终端显示器上。为了在此程序中也能体现 scanf() 函数从键盘输入信息的功能,特意在程序的结尾处使用此函数读取信息,然后通过 printf() 函数再将此信息输出。

程序的运行结果如图 16.7 所示。

写到文本文件中的内容如图 16.8 所示。

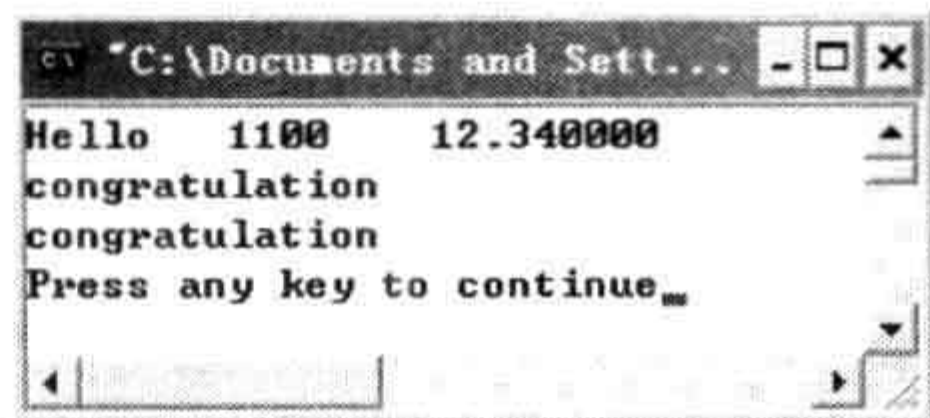


图 16.7 格式化读写函数的应用



图 16.8 写入文本文件中的数据

专家点评

这四个格式化读写函数的应用都写到了上述程序中, fscanf() 函数和 fprintf() 函数每次只能读写一个结构的元素,因此在读取磁盘文件中数据时,使用了三个 fscanf() 语句,分别读取三种不同数据类型的数据信息。

问题 285 如何判断文件的结束?

问题阐述

在文件中查找匹配的信息时,需要遍历文件中的数据信息。在遍历的过程中,如何判断文件的指针已经到了文件的结尾呢?

专家解答

1. 问题解析

在对文件的操作函数中,除了存在读写文件的函数,还有用于测试文件流是否在结尾



的函数，那就是 `feof()` 函数。

该函数的一般形式为：

```
int feof( FILE *stream );
```

其中 `stream` 是文件类型的指针。若函数的指针是指向结尾的，那么该函数返回值为 0，否则返回值为 1。该函数既可以用于检测二进制文件，又可以用来检测文本文件。

2. 实践应用

使用 `feof()` 函数测试文件流是否在结尾，若不在结尾，进入循环，读取文件中的内容，并累加文件中的字符个数。输出此个数。相应代码如下。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int count,num = 0;
    char buffer[128];
    FILE *stream;
    stream = fopen("text.txt","r");          /*打开文件*/
    while(!feof(stream))                    /*文件流不在结尾时循环*/
    {
        count = fread(buffer,sizeof(char),128,stream); /*读取字节数*/
        num += count;
    }
    printf("读取的字符个数为: %d\n",num);
    fclose(stream);
}
```

在此例中，打开的是 `text.txt` 文本文件，这个文本文件中存放的内容如图 16.9 所示。程序的运行结果如图 16.10 所示。

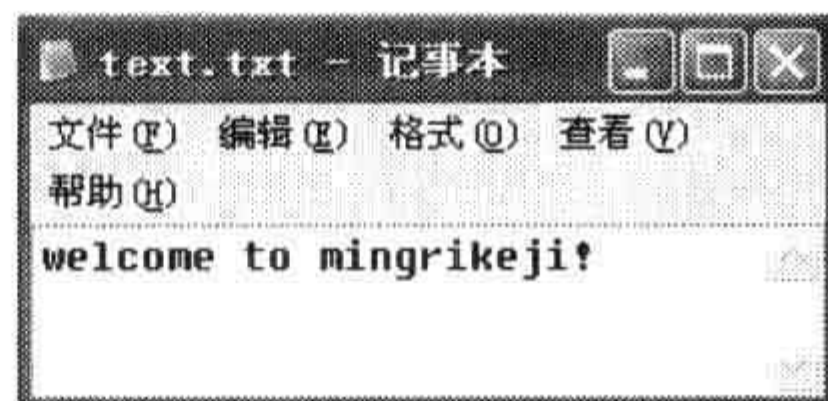


图 16.9 文本文件存放的数据

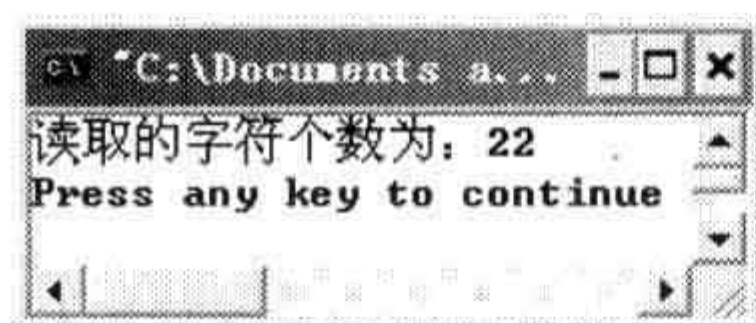


图 16.10 计算文件中的字符个数

3. 问题讨论

已知系统规定的文件结束符为 EOF，那么这个判断文件结束的符号与 `feof()` 函数测试文件结束有什么区别呢？

EOF 是 `stdio.h` 头文件中的宏定义，它的表示形式为：

```
#define EOF -1;
```

因为字符的 ASCII 码值一般不可能出现 -1，所以当读取的字符为 -1 时，表示文件结束。因此用 EOF 判断文本文件比较合适，然而对于二进制文件，使用这个 EOF 似乎就不太合



Note



适了，因为在二进制文件中是可以含有-1 的，因此系统提供了这个 `feof()` 函数用于测试二进制文件的结束。

专家点评

检测文件流是否指向结尾具有很重要的应用，例如，上述例子中的计算文件中的数据信息有多少个字符。要了解这个 `feof()` 函数不仅可以测试二进制文件是否到结尾，也可以测试文本文件是否到结尾。

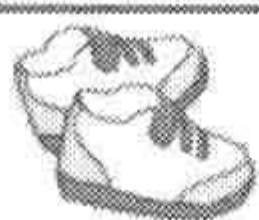


Note

第17章

图形图像处理

- ▶▶ 为什么在使用图形函数时要首先初始化图形模式?
- ▶▶ 怎样初始化图形模式?
- ▶▶ 初始化时提示“BGI Error: Graphics not initialized (use 'initgraph')”怎么办?
- ▶▶ 怎样利用C语言建立独立的图形运行程序?
- ▶▶ TC中有几个画线函数? 怎么使用?
- ▶▶ TC中有几个画矩形函数? 怎么使用?
- ▶▶ TC中有几个画圆函数? 怎么使用?
- ▶▶ 如何使用C语言填充封闭图形?
- ▶▶ TC中有几个和光标有关的函数? 怎样使用?
- ▶▶ 如何在图形模式下输出文本?
- ▶▶ 背景色、线条颜色和填充颜色有什么区别? 何时使用?
- ▶▶ 怎样记住那么多的颜色?
- ▶▶ 线条样式和填充样式都有哪些? 怎样设置?
- ▶▶ 怎样复制图形?
- ▶▶ 怎样在C语言中制作动画?



问题 286 为什么在使用图形函数时要首先初始化图形模式?

问题阐述

在 TC 中开发图形程序,为什么要先执行 `initgraph()`,这个函数是干什么用的?

专家解答

在 DOS 环境下,PC 屏幕的显示分为字符模式和图形模式两种。TC 2.0 默认工作在字符模式下。在该模式下,屏幕由 25 行 80 列(或 40 列)构成,每个栅格可以显示一个字符。由于定位函数只能定位到字符,不能定位到字符中的某一像素,这样的设置是无法完成图形绘制功能的。想要作图,屏幕必须以像素为单位定位,而不是以字符为单位,因此必须使屏幕工作在图形模式下。画图程序的基本格式如下。

```
#include<graphics.h>
main()
{
    int driver=DETECT,mode;
    initgraph(&driver,&mode,"");          /*初始化图形模式*/
    /*画图函数的使用*/
    getch();                               /*接收键盘输入,以便看清图形*/
    closegraph();                          /*关闭图形模式*/
}
```

专家点评

字符模式和图形模式是两种不同的显示模式,在字符模式下无法完成图形绘制工作。

问题 287 怎样初始化图形模式?

问题阐述

在 C 语言中,`initgraph()`函数用于初始化图形模式。初始化时,那么多参数都是干什么的?怎样设置?

专家解答

`initgraph()`函数用于初始化图形模式,其语法格式如下。

```
void far initgraph(int far *gdriver, int far *gmode,char *path);
```




其中, gdriver 和 gmode 分别表示图形驱动器和图形模式; path 是指图形驱动程序所在的目录路径。图形驱动程序枚举常量如下 (打开 graphics.h 即可找到):

```
enum graphics_drivers {
    DETECT, /*用于自动检测*/
    CGA, MCGA, EGA, EGA64, EGAMONO, IBM8514, /*1~6*/
    HERCMONO, ATT400, VGA, PC3270, /*7~10*/
    CURRENT_DRIVER = -1
};
```



Note

有关图形驱动器、图形模式的符号常数及对应的分辨率如表 17.1 所示 (打开 graphics.h 即可找到)。

表 17.1 图形驱动器、图形模式的符号常数及数值、色调、分辨率

图形驱动器 (gdriver)		图形模式 (gmode)		色 调	分辨率
符号常数	数 值	符号常数	数 值		
CGA	1	CGAC0	0	C0	320*200
CGA	1	CGAC1	1	C1	320*200
CGA	1	CGAC2	2	C2	320*200
CGA	1	CGAC3	3	C3	320*200
CGA	1	CGAHI	4	2色	640*200
MCGA	2	MCGAC0	0	C0	320*200
MCGA	2	MCGAC1	1	C1	320*200
MCGA	2	MCGAC2	2	C2	320*200
MCGA	2	MCGAC3	3	C3	320*200
MCGA	2	MCGAMED	4	2色	640*200
MCGA	2	MCGAHI	5	2色	640*480
EGA	3	EGALO	0	16色	640*200
EGA	3	EGAHI	1	16色	640*350
EGA64	4	EGA64LO	0	16色	640*200
EGA64	4	EGA64HI	1	4色	640*350
EGAMON	5	EGAMONHI	0	2色	640*350
IBM8514	6	IBM8514LO	0	256色	640*480
IBM8514	6	IBM8514HI	1	256色	1024*768
VGA	9	VGALO	0	16色	640*200
VGA	9	VGAMED	1	16色	640*350
VGA	9	VGAHI	2	16色	640*480

图形驱动程序由 Turbo C 出版商提供, 文件扩展名为.BGI。针对不同的图形适配器, 有不同的图形驱动程序可供调用。例如, 对于 EGA、VGA 图形适配器将调用驱动程序 EGAVGA.BGI。例如:

```
int driver=VGA,mode=VGAHI;
```




```
initgraph(&driver,&mode,"");
```

即将屏幕驱动程序初始化为 VGA，图形模式初始化为 VGAHI。也许有的读者会问：“这么多驱动程序，我怎么知道我的电脑是什么驱动程序呀？”

一般来说，一个显示卡支持以上多种驱动程序。此外，还可以使用 DETECT 测试你的电脑是什么驱动程序。程序写成：

```
int driver=DETECT,mode;
initgraph(&DETECT,&mode,"");
```

就可以了。

还可以用 detectgraph 函数检测，对应程序如下。

```
int driver,mode;
detectgraph(&driver,&mode)
initgraph(&DETECT,&mode,"");
```

initgraph()的第三个参数 path 是图形驱动程序的路径，即告诉程序到哪里能找到.BGI 文件。有些 TC 软件将其存放在 TC 安装目录下，有些则是在 TC 安装目录的子目录 BGI 下，自己找到 TC 的安装位置看一下就知道了。该参数设置为空白字符串时，系统首先在程序执行时的当前目录下查找，不存在时再到 C:\TC 目录下寻找，如果都找不到，则会显示如下错误信息。

BGI Error: Graphics not initialized (use 'initgraph')

专家点评

屏幕初始化是开发图形程序的第一步。各种参数设置，只要记住测试方式，基本上就可以完成所有初始化工作。大部分的图形驱动测试结果是 VGA，模式是 VGAHI，16 色，640*480 分辨率。

问题 288 初始化时提示 “BGI Error: Graphics not initialized(use 'initgraph')” 怎么办？

问题阐述

图形程序运行时，显示 “BGI Error: Graphics not initialized (use 'initgraph')”，按 Enter 键就退出了，怎么办？

专家解答

有两种情况，一种是在 TC 集成环境中出现以上错误；另一种是程序编译后，脱离 TC 环境时出现。

对于第一种情况，参见问题 287。



对于第二种情况,在软件发布后,已经没有 TC 支持了。解决方法有两种:

- ☑ 可以把图形驱动程序(.BGI)和自己的程序放在同一文件夹下发布,将 initgraph 函数的第三个参数设置为空白字符串。
- ☑ 参见问题 289。

专家点评

这是图形程序中常见的错误,必须解决,如果这一步解决不了,那什么图也作不了。

问题 289 怎样利用 C 语言建立独立的图形运行程序?

问题阐述

自己开发的 TC 图形程序运行时,要连同图形驱动程序(.BGI 文件)同时发布,才能正常运行。那么,怎样把.BGI 文件编译到.exe 文件中去呢?

专家解答

Turbo C 中规定按照下述步骤(这里以 EGA、VGA 显示器为例)进行操作:

(1) 在 C:\TC 子目录下输入命令“BGI\OBJ EGAVGA”,将驱动程序 EGAVGA.BGI 转换成 EGAVGA.OBJ 的目标文件。

(2) 在 C:\TC 子目录下输入命令“TLIB LIB\GRAPHICS.LIB+EGAVGA”,将 EGAVGA.OBJ 的目标模块装到 GRAPHICS.LIB 库文件中。

(3) 在程序中 initgraph()函数调用之前加上一句“registerbgidriver(EGAVGA_driver”,告诉连接程序在连接时把 EGAVGA 的驱动程序装入到用户的执行程序中。经过上面的处理,编译链接后的执行程序可在任何目录或其他兼容机上运行。假设已做了前两个步骤,加入 registerbgidriver()函数的位置如下。

```
#include<graphics.h>
#include<stdio.h>
main()
{
    int driver=DETECT,mode;
    registerbgidriver(EGAVGA_driver);
    initgraph(&driver,&mode,"");
    /*画图函数的使用*/
    getch();
    closegraph();
}
```

这样,编译链接后产生的执行程序可独立运行。如不初始化成 EGA 或 CGA 分辨率,而想初始化为 CGA 分辨率,则只需将上述步骤中有 EGAVGA 的地方用 CGA 代替即可。



Note



专家点评

将图形驱动程序编译到自己的程序中去,可开发出独立 TC 开发环境的图形运行程序,就不用在自己的程序之外再带上个.BGI 文件了。



Note

问题 290 TC 中有几个画线函数? 如何使用?

问题阐述

C 语言中画线的函数好像不止 line() 一个,那么除了 line(), 还有哪些画线函数? 如何使用?

专家解答

TC 中有 3 种画线的函数,共语法格式如下。

void far line(int x0, int y0, int x1, int y1);	/*画一条从点(x0, y0)到(x1, y1)的直线*/
void far lineto(int x, int y);	/*画一条从当前光标到点(x, y)的直线*/
void far linerel(int dx, int dy);	/*画一条从当前光标(x, y)到由相对增量确定的点(x+dx, y+dy)的直线*/

提示:

当前光标即当前作图点的位置

例如,画一个正三角形,3 条边分别用 3 种方法来画。首先用 line()画一条底边,直接指定两个端点;然后用 moveto()把当前光标移到底边右端点;再用 linerel()画右上边,画完后的当前光标直接在顶点处;最后用 lineto()画左上边。

```
#include<stdio.h>
#include<graphics.h>          /*引用图形库头文件*/
#include<math.h>
main()
{
    int driver=DETECT,mode;
    int x,y;
    initgraph(&driver,&mode,"");    /*初始化图形模式*/
    line(100,400,300,400);          /*画一条底边*/
    moveto(300,400);                /*将光标移到底边右端点(300,400)*/
    y=200*sin(60*3.1416/180);        /*计算等边三角形上顶点 x,y 增加值*/
    x=200*cos(60*3.1416/180);
    linerel(-x,-y);                 /*从右下端点到上顶点画一条线*/
    lineto(100,400);                /*从上顶点到左下端点画一条线*/
    getch();
```




```
closegraph();           /*关闭图形模式*/
}
```

程序进行结果如图 17.1 所示。

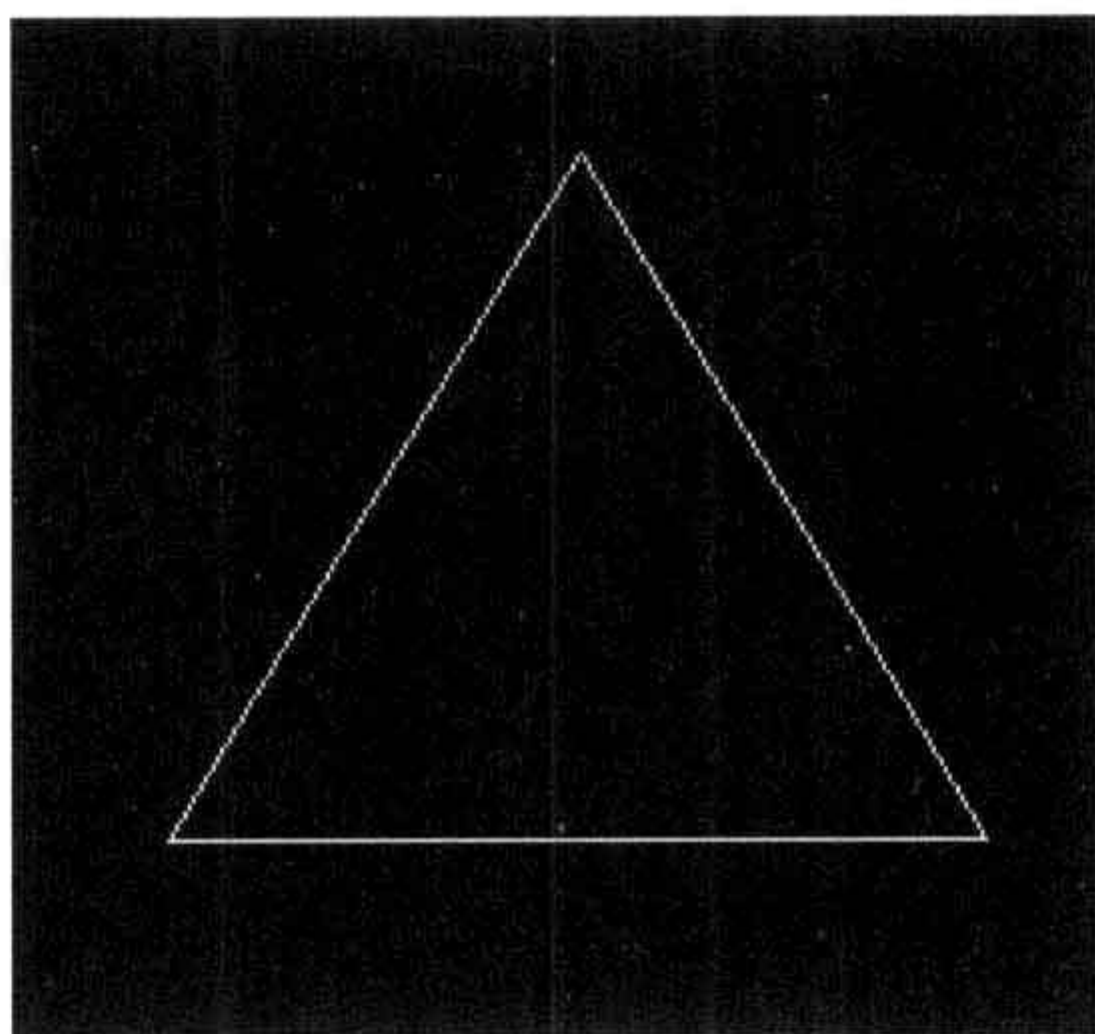


图 17.1 用 3 种画线函数画的正三角形



Note

专家点评

3 种画线函数指定直线两个端点的方式不同, 灵活应用可以减少计算和绘图工作量。

问题 291 TC 中有几个画矩形函数? 怎么使用?

问题阐述

TC 中有哪些画矩形的函数? 怎么使用?

专家解答

TC 中有 5 个画矩形函数, 其语法格式如下。

```
void far rectangle(int left, int top, int right, int bottom);    /*绘制一个矩形边框*/
void far bar(int x1, int y1, int x2, int y2);                  /*绘制一个填充的矩形窗口*/
void far bar3d(int x1, int y1, int x2, int y2, int depth, int topflag);
    /*当 topflag 为非零时, 绘制出一个三维的长方体; 当 topflag 为 0 时, 三维图形不封顶*/
void fillpoly(int numpoints, int far *polypoints);
    /*绘制一个填充多边形, 边数为 numpoints, 顶点在数组 polypoints 中, 该数组每相邻两
    个数是一个点的坐标(x,y)*/
void drawpoly(int numpoints, int far *polypoints);
    /*绘制一个填充的多边形。其中参数 numpoint、polypoints 的含义与 fillpoly()中的相同。
    要绘制封闭五边形, 要有 6 个点, 第六点坐标与第一点坐标相同*/
```

下面程序演示了各函数的应用, 代码如下。

```
#include<stdio.h>
#include<graphics.h>           /*引用图形库头文件*/
#include<math.h>
```




Note

```
#define PI 3.1415926
main()
{
    int driver=DETECT,mode;
    int i;
    int point[12];
    initgraph(&driver,&mode,"");          /*初始化图形模式*/
    setcolor(YELLOW);                     /*线条颜色*/
    rectangle(350,100,450,200);           /*画矩形边框*/
    setfillstyle(SOLID_FILL,GREEN);       /*填充样式*/
    bar3d(100,100,200,200,50,1);         /*三维长方体*/
    bar(100,300,200,400);                 /*填充矩形*/
    for(i=0;i<5;i++)                      /*计算填充正五边形顶点*/
    {
        point[2*i]=sin((180+72*i)/180.0*PI)*50+300;
        point[2*i+1]=cos((180+72*i)/180.0*PI)*50+360;
    }
    fillpoly(5,point);                    /*绘制填充正五边形*/
    for(i=0;i<5;i++)                      /*计算五角星顶点*/
    {
        point[2*i]=sin((180+144*i)/180.0*PI)*50+450;
        point[2*i+1]=cos((180+144*i)/180.0*PI)*50+360;
    }
    point[10]=point[0];                   /*第六点坐标与第一点坐标相同*/
    point[11]=point[1];
    drawpoly(6,point);                    /*画五角星*/
    getch();
    closegraph();                          /*关闭图形模式*/
}
```

程序运行结果如图 17.2 所示。

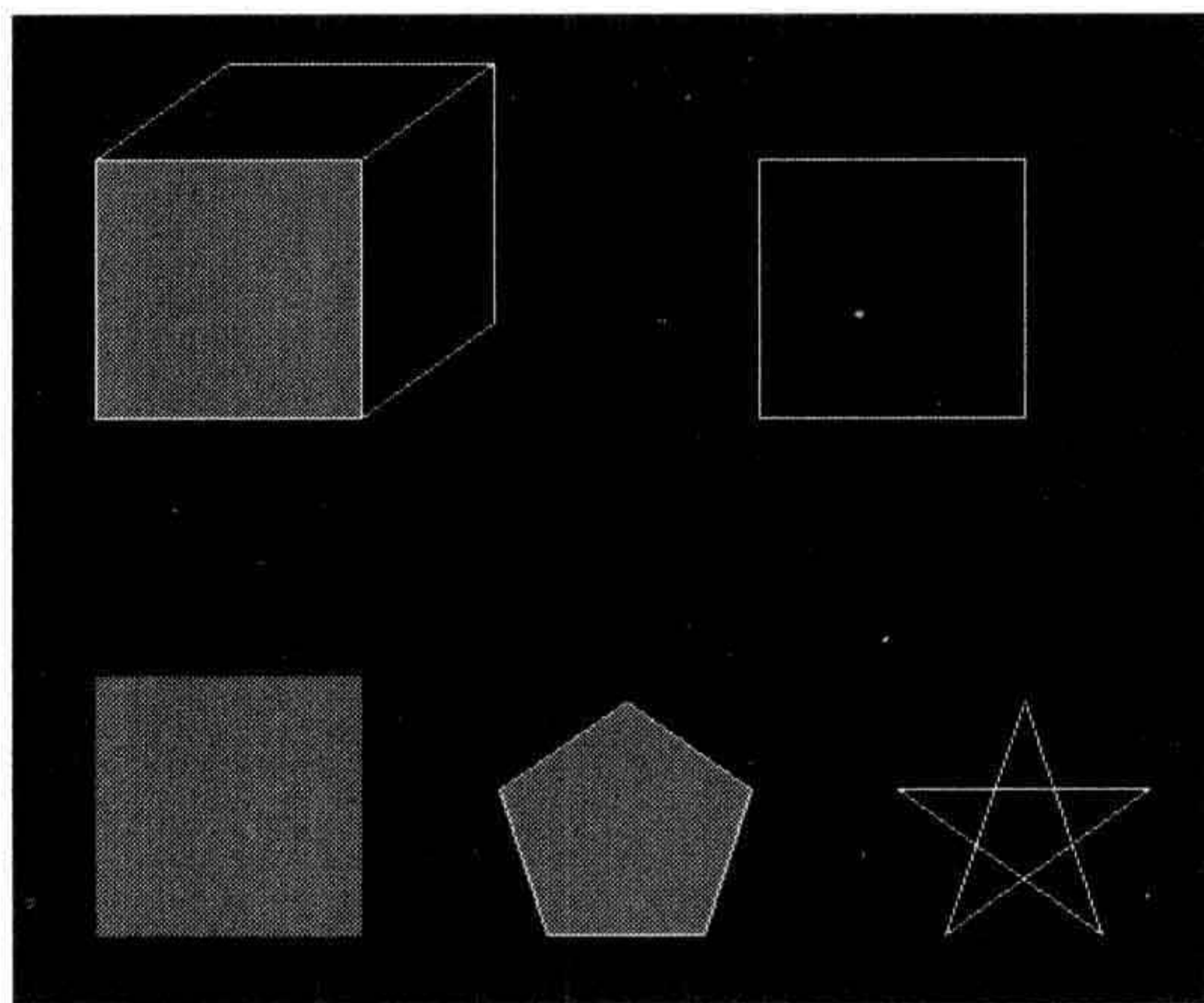


图 17.2 矩形（含多边形）函数的使用



专家点评

矩形是最常用的图形，可以由矩形构成其他复杂图形，因此一定熟练掌握。

问题 292 TC 中有几个画圆函数？怎么使用？



Note

问题阐述

TC 中有几个画圆函数？怎么使用？

专家解答

TC 中有 6 个画圆函数，其语法格式如下。

```
void circle(int x, int y, int radius);          /*以(x,y)为圆心、radius 为半径画圆，没有填充*/
void arc(int x, int y, int stangle, int endangle,int radius);
    /*画一个以(x, y)为圆心、radius 为半径、stangle 为起始角、endangle 为终止角的圆扇形，
    没有填充*/
void ellipse(int x, int y, int stangle, int endangle,int xradius, int yradius);
    /*画一个以(x, y)为圆心分别以 xradius 为 x 轴半径、yradius 为 y 轴半径、stangle 为起始角，
    endangle 为终止角的椭圆扇形，没有填充*/
void far pieslice(int x,int y,int stangle,int endangle,int radius);
    /*画一个以(x, y)为圆心、radius 为半径、stangle 为起始角度、endangle 为终止角度的正圆
    扇形，再按规定方式填充。当 stangle=0、endangle=360 时变成一个实心圆，并在圆内
    从圆点沿 x 轴正向画一条半径*/
void far sector(int x, int y,int stanle,intendangle,int xradius, int yradius);
    /*画一个以(x, y)为圆心、xradius yradius 为 y 轴半径、stangle 为起始角、endangle 为终止
    角的椭圆扇形，再按规定方式填充*/
void fillellipse( int x, int y, int xradius, int yradius );
    /*画一个以(x, y)为圆心_xradius 为 x 轴半径、yradius 为 y 轴半径的椭圆扇形，并填充*/
```

下面程序演示了各画圆函数的应用，从中可以看到它们各自的特点。

```
#include<stdio.h>
#include<graphics.h>          /*引用图形库头文件*/
#include<math.h>
#define PI 3.1415926
main()
{
    int driver=DETECT,mode;
    initgraph(&driver,&mode,"");
    setcolor(BREEN);          /*设置线条颜色为绿色*/
    setfillstyle(SOLID_FILL,WHITE); /*将填充样式设置为白色实心*/
    circle(100,100,50);        /*画空心圆*/
    arc(300,100,0,270,50);     /*画空心部分圆，即弧线*/
```




Note

```

ellipse(500,100,0,360,80,50);           /*空心椭圆*/
pieslice(100,300,0,270,50);              /*实心正圆扇形*/
sector(300,300,0,270,80,50);             /*实心椭圆扇形*/
fillellipse(500,300,80,50);              /*实心椭圆*/
getch();
closegraph();                             /*关闭图形模式*/
}

```

程序运行结果如图 17.3 所示。

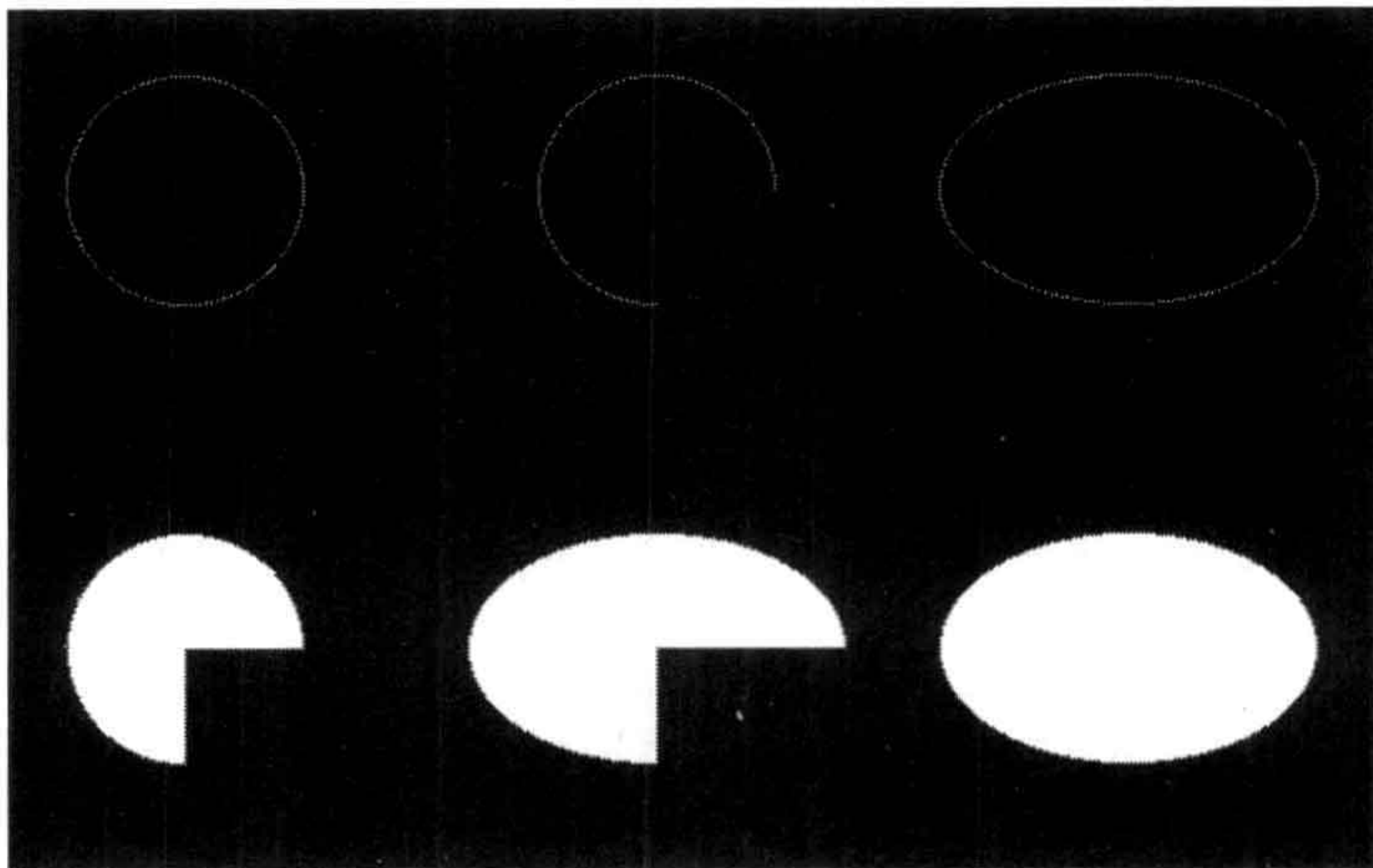


图 17.3 各种圆的画法

专家点评

这些画圆的函数功能各异，有的画部分圆，有的只画边界线，有的还可以画椭圆。用户可根据实际情况自行选用。

问题 293 如何使用 C 语言填充封闭图形？

问题阐述

TC 中的函数有些只能画边线，不能填充。如要对封闭图形（如两个区域的交集）进行填充，那么怎么实现呢？

专家解答

填充就是用指定的颜色和图案填满一个封闭图形。

TC 提供了一个可对任意封闭图形填充的函数，即 `floodfill()`。其调用格式如下：

```
void far floodfill(int x, int y, int border);
```

其中，`x`、`y` 为封闭图形内的任意一点；`border` 为边界的颜色，也就是封闭图形轮廓的



颜色。

调用了该函数后，将用指定的颜色和图案填满整个封闭图形。

注意：

- (1) 如果 x 或 y 取在边界上，则不进行填充。
- (2) 如果不是封闭图形，则填充会从没有封闭的地方溢出去，填满其他地方。
- (3) 如果 x 或 y 在图形外面，则填充封闭图形外的屏幕区域。
- (4) 由 `border` 指定的颜色值必须与图形轮廓的颜色值相同，但填充色可选任意颜色。
- (5) 填充颜色和图案由 `setfillstyle()` 函数决定，具体参见问题 298。



Note

下面举例说明 `floodfill()` 函数的用法。先画一个五角星边界线，形成中心 1 个填充区和外围 5 个填充区，然后再填充这 6 个区域。代码如下：

```
#include<stdio.h>
#include<graphics.h>           /*引用图形库头文件*/
#include<math.h>
#define PI 3.1415926
main()
{
    int driver=DETECT,mode;
    int i,maxx,maxy;
    int point[12];
    initgraph(&driver,&mode,"");    /*初始化图形模式*/
    maxx=getmaxx();
    maxy=getmaxy();
    setcolor(YELLOW);              /*线条颜色*/
    setfillstyle(SOLID_FILL,YELLOW); /*填充样式*/
    for(i=0;i<5;i++)              /*计算五角星顶点*/
    {
        point[2*i]=sin((180+144*i)/180.0*PI)*100+maxx/2;
        point[2*i+1]=cos((180+144*i)/180.0*PI)*100+maxy/2;
    }
    point[10]=point[0];           /*第六点坐标与第一点坐标相同*/
    point[11]=point[1];
    drawpoly(6,point);            /*画五角星*/
    floodfill(maxx/2,maxy/2,YELLOW); /*填充中心点*/
    for(i=0;i<5;i++)              /*填充外围 5 个三角形*/
        floodfill(sin((180+144*i)/180.0*PI)*50+maxx/2,
            cos((180+144*i)/180.0*PI)*50+maxy/2,YELLOW);
    getch();
    closegraph();                 /*关闭图形模式*/
}
```

程序运行结果如图 17.4 所示。



Note

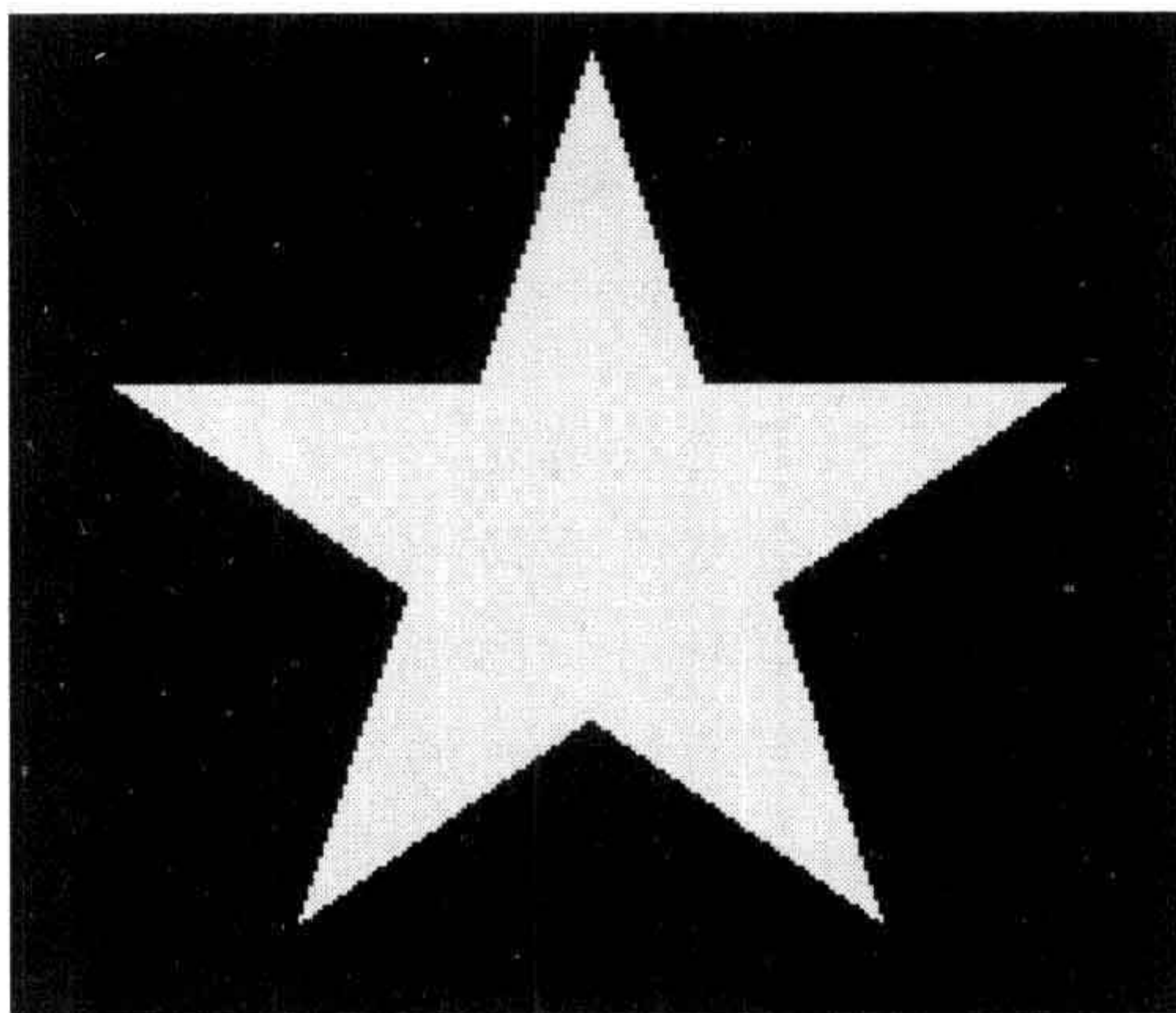


图 17.4 填充函数的使用

专家点评

fillflood()函数能够实现对封闭图形进行填充的功能。

问题 294 TC 中有几个和光标有关的函数？怎样使用？

问题阐述

TC 中哪几个作图函数是在当前光标处进行操作的？怎样使用呢？

专家解答

TC 中共有 6 个函数和光标有关，其语法格式如下。

int far getmaxx(void);	/*返回 x 轴的最大值*/
int far getmaxy(void);	/*返回 y 轴的最大值*/
int far getx(void);	/*返回当前光标在 x 轴的位置*/
void far gety(void);	/*返回当前光标在 y 轴的位置*/
void far moveto(int x, int y);	/*移动光标到(x, y)点，不是画点*/
void far moverel(int dx, int dy);	/*将光标从当前位置(x, y)移动到(x+dx, y+dy)的位置，移动过程中不画点*/

不同图形驱动程序、不同显示模式下的屏幕分辨率是不同的，具体值可以参阅问题 287。在程序中，可以使用 getmaxx()和 getmaxy()函数获取屏幕最大 x,y 坐标。例如，在屏幕中心画一水平直线，可写为：

```
line(0,getmaxy()/2,getmaxx(),getmaxy()/2);
```

有些函数执行时会改变当前光标位置，如 lineto()、linerel()、moveto()、moverel()、outtext()、outtextxy()等，而 getx()、gety()函数用于获取当前光标位置。



专家点评

光标操作虽然是作图的辅助功能，但如果不会灵活控制光标，就没法控制图形位置。

问题 295 如何在图形模式下输出文本？



Note

问题阐述

在图形模式下，文本是否也可以图形化，显示出与文本方式下不同的效果？

专家解答

在图形模式下，可以使用标准输出函数，如 `printf()`、`puts()`、`putchar()` 函数输出文本到屏幕。除此之外，Turbo C 2.0 还提供了一些专门用于在图形模式下输出文本的函数。

1. 文本输出函数

```
void far outtext(char far *textstring);           /*在当前光标处输出 textstring 所指的字符串*/
void far outtextxy(int x, int y, char far *textstring); /*在(x,y)处输出 textstring 所指的字符串*/
```

2. 字符串格式化函数

```
int sprintf(char *str, char *format, variable-list);
/*它与 printf() 函数的不同之处是 printf() 函数将格式化结果输出到屏幕，sprintf() 将格式化结果保存在 str 指向的字符串中，返回值是写入的字符个数*/
```

例如：

```
sprintf(s, "current cursor %d,%d", getx(), gety());
```

这里 `s` 应是字符串指针。

3. 用于对文本输出时的对齐方式进行设置的函数

```
void far settexjustfy(int horiz, int vert);
```

对于 `outtextxy()` 和 `outtext()` 函数所输出的字符串，其中哪个点对应于当前坐标 `(x,y)`，在 Turbo C 2.0 中是有规定的。

- ☒ 如果水平对齐方式设置为左对齐，那么文本左边界对齐 `(x,y)`，输出从当前 `(x,y)` 开始，向右输出。
- ☒ 如果水平对齐方式设置为右对齐，那么文本右边界对齐 `(x,y)`，输出从当前 `(x,y)` 开始，所有输出都位于当前点 `(x,y)` 左侧。
- ☒ 如果水平对齐方式设置为中间对齐，那么输出后的字符串中间点在 `x` 处。

垂直对齐方式与上述水平对齐类似。

在 `settexjustfy()` 函数中，第一个参数 `horiz` 指定水平对齐方式；第二个参数 `vert` 指定垂直对齐方式。

有关参数 `horiz` 和 `vert` 的取值说明如下。



Note

```
enum text_just{                                /*水平、垂直对齐方式*/
    LEFT_TEXT    = 0,
    CENTER_TEXT  = 1,
    RIGHT_TEXT   = 2,
    BOTTOM_TEXT  = 0,
    /*CENTER_TEXT = 1,
    TOP_TEXT     = 2
};
```

4. 用于对字体、字号、输出方向进行设置的函数

```
void far settextstyle(int font, int direction, int charsize); /*设置输出字符的字体 font、输出方向
                                                                direction 和字号 charsize*/
```

Turbo C 2.0 对该函数中各个参数的规定如表 17.2~表 17.4 所示。

表 17.2 font 的取值

符号常数	数值	含义
DEFAULT_FONT	0	8*8点阵字（默认值）
TRIPLEX_FONT	1	三倍笔划字体
SMALL_FONT	2	小号笔划字体
SANS_SERIF_FONT	3	无衬线笔划字体
GOTHIC_FONT	4	黑体笔划字

表 17.3 direction 的取值

符号常数	数值	含义
HORIZ_DIR	0	从左到右
VERT_DIR	1	从底到顶

表 17.4 charsize 的取值

参数	数值	含义
charsize	1~10	每一个字号相当于8像素，即1号8*8像素，2号16*16像素，以此类推

还可以用 USER_CHAR_SIZE=0 表示用户定义的字符大小。

下面举例说明图形模式下有关文本输出和字体、字号等设置函数的用法，代码如下。

```
#include<stdio.h>
#include<graphics.h>                                /*引用图形库头文件*/
#include<math.h>
main()
{
    int driver=DETECT, mode;
    char s[30];
    initgraph(&driver, &mode, "");
    setbkcolor(BLUE);                                /*设置背景色*/
    setfillstyle(SOLID_FILL, GREEN);                 /*绿色实心填充*/
```




```

setcolor(YELLOW);
rectangle(100, 100, 539, 379);
floodfill(150, 150, YELLOW);
setcolor(LIGHTRED);
settextstyle(DEFAULT_FONT, HORIZ_DIR, 3);
outtextxy(140, 220, "MingRi SoftWare");
setcolor(WHITE);
settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 1);
outtextxy(120, 120, "Ming Ri");
setcolor(YELLOW);
settextstyle(SMALL_FONT, HORIZ_DIR, 8);
sprintf(s, "screen maxx=%d,maxy=%d", getmaxx(),getmaxy());
outtextxy(130, 300, s);
setcolor(BLUE);
settextstyle(4, HORIZ_DIR, 5);
outtextxy(170, 150, "book publish");
getch();
closegraph();
return 0;
}

```

/*设置画线颜色*/
/*画矩形输出面板*/
/*设置填充色*/
/*文字显示颜色*/
/*默认字体, 水平, 3号*/
/*输出文字*/
/*设置文字显示颜色*/
/*无衬线笔画字体, 水平, 1号*/
/*将数字转换为字符串*/
/*指定位置输出字符串*/



Note

程序运行结果如图 17.5 所示。

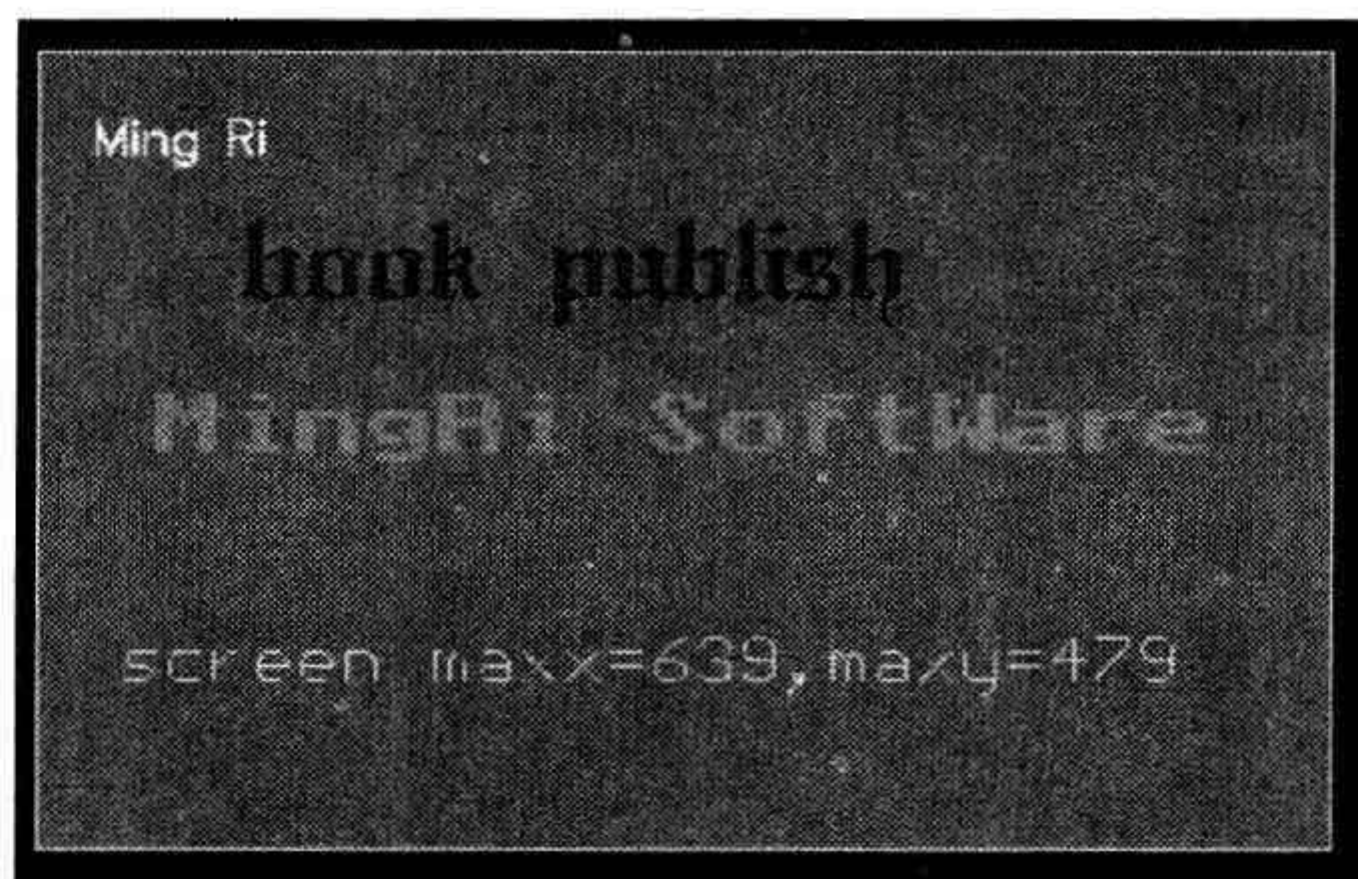


图 17.5 在图形模式下输出文本

5. 用于对文本字符大小进行设置的函数

```
void far setusercharsize(int mulx, int divx, int muly, int divy); /*为矢量字体改变字符宽度和高度*/
```

它是在 settextstyle()已设置的字号基础上增加高度、宽度或减少高度、宽度。

注意:

点阵字体和矢量字体中, 只在矢量字体才能用 setusercharsize()改变字号大小, 点阵字体只能用 settextstyle()设置字号。

在 settextstyle()函数已经设定的字号基础上, mulx 可以将原来字号在水平方向上增加 mulx 倍、muly 可以将原来字号在垂直方向上增加 muly 倍、divx 可以将原来字号在水平方向上缩小至 1/divx、divy 可以将原来字号在垂直方向上缩小至 1/divy。



Note

```

#include<stdio.h>
#include<graphics.h>                                /*引用图形库头文件*/
main()
{ int driver,mode;
  detectgraph(&driver,&mode);                        /*检测图形驱动程序, 图形模式*/
  initgraph(&driver,&mode,"d:\\tc");                /*初始化图形模式*/
  settextstyle(TRIPLEX_FONT,HORIZ_DIR,3);          /*设置文本样式, 矢量字体, 水平方向,
                                                    字号 3 号*/
  moveto(10,200);                                    /*光标移到(10,200)*/
  outtext("Normal ");                               /*输出 Normal*/
  setusercharsize(1,1,2,1);                         /*字的高度*2*/
  outtext("Height y*2 ");                          /*输出*/
  setusercharsize(2,1,1,1);                         /*字的宽度*2*/
  outtext("Wide x*2");                             /*输出*/
  getch();                                           /*等待键盘输入, 以便看清图形*/
  closegraph();                                     /*关闭图形模式*/
}

```

程序运行结果如图 17.6 所示。



图 17.6 自定义字形

专家点评

在图形模式下, 可以对文本的字体、字号、位置、对齐方式进行设置。

问题 296 背景色、线条颜色和填充颜色有什么区别? 何时使用?

问题阐述

背景色、线条颜色和填充颜色, 这几种颜色有什么区别? 什么时候使用?

专家解答

- ☑ 背景色: 是整个屏幕的底色, 设置之后, 屏幕空白区域都变成该颜色。
- ☑ 线条颜色: 是画线时所用的颜色。文字输出也用线条颜色。
- ☑ 填充颜色: 是封闭图形边界线以内的颜色。

**注意:**

我们画的图形分为边界线和边界以内的填充部分,如圆、矩形,可以明显地分出边界部分和填充部分。

有些图形函数只能画边界线,如 `rectangle()`、`drawpoly()`、`circle()`、`arc()`、`ellipseline()`、`lineto()`、`linerel()`。对于这些函数,填充颜色不起作用。另外还有一些函数,它们既可画边界又能填充内部,如 `bar()`、`fillpoly()`、`pieslice()`、`sector()`、`fillellipse()`。对于这些函数,线条颜色、填充颜色都起作用。

想要设置背景色、线条颜色和填充颜色时,可以用如下3个函数来实现。

```
void far setbkcolor(int color);           /*用于设置背景色*/
void far setcolor(int color);             /*用于设置画线颜色*/
void far setfillpattern(char far *upattern, int color) /*用于设置填充颜色*/
void far setfillstyle(int pattern, int color); /*用于设置填充颜色*/
```

提示:

这两个函数关于线条样式和填充样式部分的详细用法参见问题298。

专家点评

以上设置颜色的函数只对它以后的画图语句起作用,也就是要先设置线条颜色,填充颜色,再去画图。

问题 297 怎样记住那么多的颜色?

问题阐述

赤、橙、黄、绿、青、蓝、紫,如此之多的颜色,数字不好记,英文看程序还可以,直接写也不好写。那么怎样记住那么多的颜色呢?

专家解答

颜色枚举值如下:

```
enum COLORS {
    BLACK,           /*0 黑*/
    BLUE,            /*1 蓝*/
    GREEN,           /*2 绿*/
    CYAN,            /*3 青*/
    RED,             /*4 红*/
    MAGENTA,         /*5 洋红*/
    BROWN,          /*6 棕*/
    LIGHTGRAY,       /*7 淡灰*/
    DARKGRAY,        /*8 深灰*/
}
```



Note



Note

```

LIGHTBLUE,          /*9 淡蓝*/
LIGHTGREEN,         /*10 淡绿*/
LIGHTCYAN,          /*11 淡青*/
LIGHTRED,           /*12 淡红*/
LIGHTMAGENTA,       /*13 淡洋红*/
YELLOW,            /*14 黄*/
WHITE              /*15 白*/
};

```

上述的枚举值与相应的数值等价，二者可以互换。例如，设定蓝色背景可以使用 `textbackground(1)`，也可以使用 `textbackground(BLUE)`，两者没有任何区别，只不过后者比较容易记忆，一看就知道是蓝色。

如果想记住那些数字，这里有一个小技巧可供参考。

可以这样想，颜色是用 4 位二进制数表示的，记为：

亮红绿蓝

如 0001 即蓝色，因为第四位蓝为 1，其他为 0，对应十进制为 1。

如 0010 即绿色，因为第三位绿为 1，其他为 0，对应十进制为 2。

如 0100 即红色，因为第二位红为 1，其他为 0，对应十进制为 4。

如 0011 即绿蓝混合色，也就是青色，因为第三位绿、第四位蓝为 1，其他为 0，对应十进制为 3。

如 0010 为绿色，1010 为亮绿色，因第一位表示对应颜色高亮显示。

什么颜色都没有是黑色，所有颜色都有是白色。

专家点评

用两种方式表示的颜色其实都不难记，枚举值那几个英语单词也都是最常用的，如果这几个颜色都记不住，那么多图形函数的函数名怎么记呀？

问题 298 线条样式和填充样式都有哪些？怎样设置？

问题阐述

线条样式和填充样式都有哪些？该怎样设置？

专家解答

在画图前，要先设置线条样式和填充样式，它们直接决定图形的显示效果。

1. 线条样式

(1) `setlinestyle()` 函数。

```
void far setlinestyle(int linestyle, unsigned upattern, int thickness);
```

```
/*指定画线的样式 (linestyle 或 upatttern) 和宽度 (thickness) */
```




线条样式常量值如下:

- ☑ SOLID_LINE=0: 实线。
- ☑ DOTTED_LINE=1: 点线。
- ☑ CENTER_LINE=2: 中心线。
- ☑ DASHED_LINE=3: 点画线。
- ☑ USERBIT_LINE=4: 用户定义线。

画线宽度常量值如下:

- ☑ NORM_WIDTH=1: 一点宽。
- ☑ THICK_WIDTH=3: 三点宽。

在没有对线条特性进行设定之前, TC 用其默认值, 即一点宽的实线。

对于 upattern, 只有 linestyle 选择 USERBIT_LINE 时才有意义 (选择其他线型, upattern 取 0 即可)。upattern 的 16 位二进制数的每一位代表 1 像素, 其值为 1 显示, 为 0 则不显示。

(2) getlinesettings()函数。

```
void far getlinesettings(struct linesettingstypefar *lineinfo);
/*将有关线条的信息存放到由 lineinfo 指向的结构中*/
```

其中 linesettingstype 的结构如下:

```
struct linesettingstype
{
    int linestyle;
    unsigned upattern;
    int thickness;
}
```

例如, 下面两句程序可以读出当前线条的特性。

```
struct linesettingstype *info;
getlinesettings(info);
```

(3) setwritemode()函数。

```
void far setwritemode(int mode); /*用于指定画线的方式*/
```

如果 mode=0, 则表示画线时将所画位置的原来信息覆盖 (这是 TC 的默认方式)。如果 mode=1, 则表示画线时用当前特性的线与所画之处原有的线进行异或 (XOR) 操作, 实际上画出的线是原有线与当前指定的线进行异或后的结果。因此, 当线的特性不变, 进行两次画线操作相当于没有画线。

有关线型设定和画线函数的实例程序如下。

```
#include<stdio.h>
#include<alloc.h>
#include<graphics.h>
main()
/*引用图形库头文件*/
```




Note

```
{
    int driver,mode,i,j;
    struct linesettingstype lineset;
    detectgraph(&driver,&mode);          /*检测图形驱动程序, 图形模式*/
    initgraph(&driver,&mode,"d:\\tc");    /*初始化图形模式*/
    for(i=0;i<4;i++)
        for(j=0;j<2;j++)
        {
            setlinestyle(i,0,2*j+1);      /*设置线条样式*/
            line(100,100+(i*4+j)*20,300,100+(i*4+j)*20); /*画线*/
        }
    setlinestyle(4,0xb77f,1);            /*1011 0111 0111 1111*/
    line(100,400,300,400);
    getch();                             /*等待键盘输入, 以便看清图形*/
    closegraph();                         /*关闭图形模式*/
}
```

程序运行结果如图 17.7 所示。

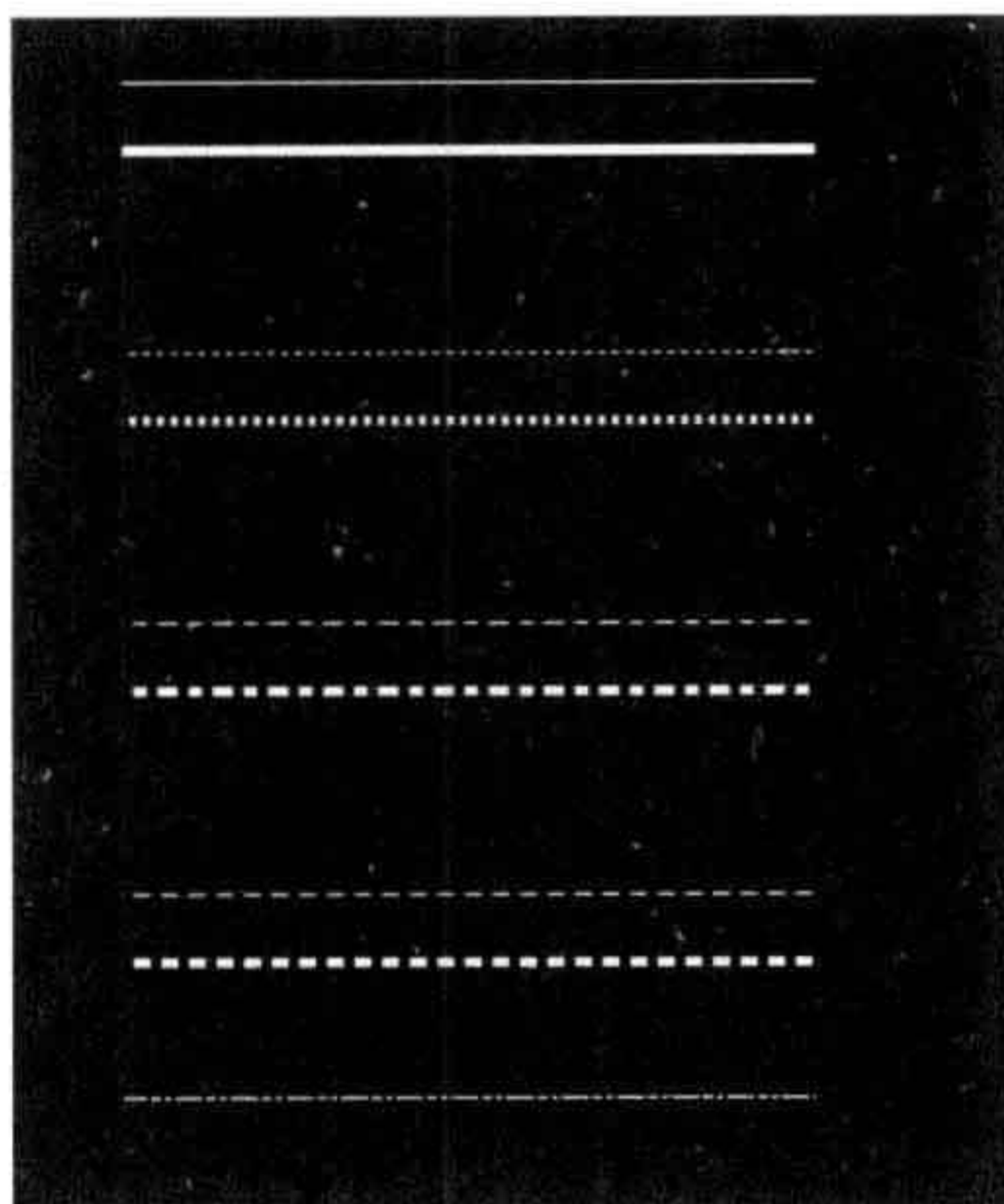


图 17.7 直线样式

2. 设置填充样式

(1) setfillstyle()函数。

```
void far setfillstyle(int pattern, int color);    /*设置填充模式和填充颜色*/
```

填充模式 patternr 的枚举定义如下。

```
enum fill_patterns {
    EMPTY_FILL,          /*0 背景颜色填充*/
    SOLID_FILL,           /*1 实心填充*/
    LINE_FILL,            /*2---填充 1*/
    LTSLASH_FILL,         /*3///填充*/
    SLASH_FILL,           /*4 粗///填充*/
    BKSLASH_FILL,         /*5 粗\\填充*/
    LTBKSLASH_FILL,       /*6\\填充*/
}
```




```

HATCH_FILL,           /*7 直方网格填充*/
XHATCH_FILL,          /*8 斜网格填充*/
INTERLEAVE_FILL,      /*9 间隔点填充*/
WIDE_DOT_FILL,        /*10 密集点填充*/
CLOSE_DOT_FILL,       /*11 密集点填充*/
USER_FILL             /*12 用户定义*/
};

```

(2) setfillpattern()函数。

```
void far setfillpattern(char far *upattern, int color);
```

函数的功能是选择用户定义的填充模式，与 setfillstyle() 功能相近，后者可以设置系统预定义的填充样式。setfillpattern 的样式定义方法，用 8*8 的点阵说明，C 语言中一个字符占一个字节，8 位二进制，可构成 8*8 点阵的一行，8 个字符即完整 8*8 点阵。

(3) getfillsettings()函数。

```
void getfillsettings(struct fillsettingstype far *fillinfo);
```

函数的功能是获取系统预定义的填充模式和填充颜色放入 fillinfo 结构中。

```

struct fillsettingstype
{
    int pattern;           /*现行填充模式*/
    int color;            /*现行填充模式*/
};

```

(4) getfillpattern()函数。

```
void getfillpattern(char far *pattern);
```

函数的功能是获取用户定义的填充模式。

```

#include<stdio.h>
#include<graphics.h>           /*引用图形库头文件*/
main()
{
    int driver,mode,i,j;
    char buf[100];
    char pattern[8]={0x80, 0x80, 0x80, 0xff, 0xff, 0x00, 0x00, 0x00};
    detectgraph(&driver,&mode); /*检测图形驱动程序，图形模式*/
    initgraph(&driver,&mode,"d:\\tc"); /*初始化图形模式*/
    setttextjustify(CENTER_TEXT,CENTER_TEXT); /*文字对齐方式*/
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
        {
            setfillstyle(i*4+j,3); /*值为 0,1,2,3…….顺序设置不同样式
                                     值，用于填充圆*/
            fillellipse(50+j*100,50+i*100,40,40); /*在屏幕上三行四列方式画 12 个圆*/
        }
}

```



Note



```

        sprintf(buf,"pattern=%d",i*4+j);
        outtextxy(50+j*100,50+i*100+50,buf);    /*在圆下方显示对应填充样式*/
    }
    setfillpattern(pattern,3);
    fillellipse(50,350,40,40);
    outtextxy(50,410,"user define");
    getch();    /*等待键盘输入，以便看清图形*/
    closegraph();    /*关闭图形模式*/
}

```

程序运行结果如图 17.8 所示。

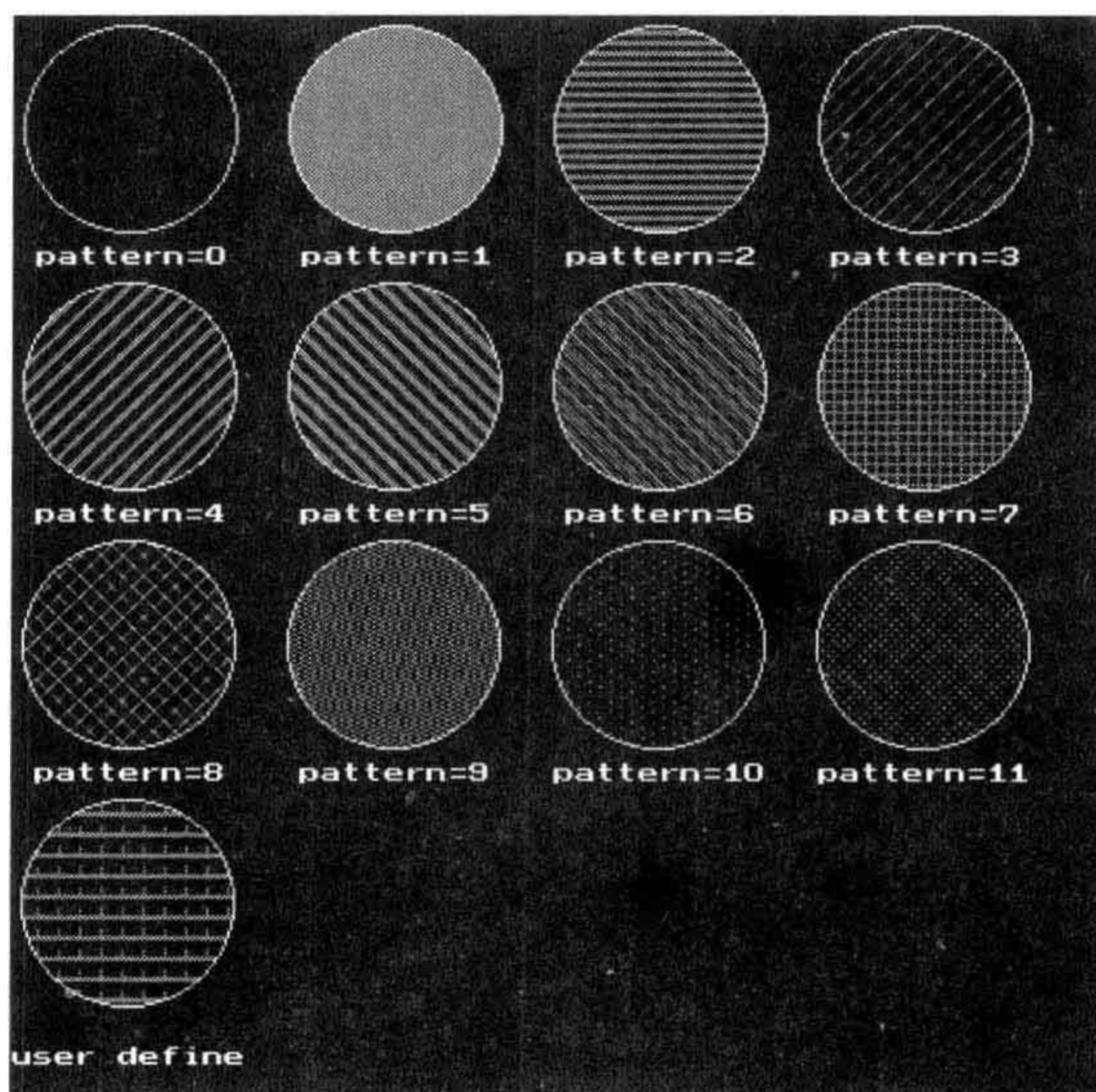


图 17.8 填充样式

专家点评

在不需要特殊效果的时候，不设置线条样式和填充样式，即直接使用其默认样式——实线和实心填充，就可以得到我们所需要的效果。

问题 299 怎样复制图形？

问题阐述

C 语言中如何实现把屏幕某一处的图形复制到另一处？

专家解答

复制图形要用到以下 3 个函数：



```
void far getimage(int xl,int yl, int x2,int y2,void far *mapbuf);
void far putimage(int x,int,y,void * mapbuf, int op);
unsigned far imagesize(int xl,int yl,int x2,int y2);
```

这 3 个函数用于将屏幕上的图像复制到内存,然后再将内存中的图像送回到屏幕上。首先通过函数 `imagesize()` 测试要保存左上角为(xl,yl), 右上角为(x2,y2)的图形屏幕区域内的全部内容需多少个字节,然后再给 `mapbuf` 分配一个所测数字字节内存空间的指针。通过调用 `getimage()`函数就可将该区域内的图像保存在内存中,需要时可用 `putimage()`函数将该图像输出到左上角为点(x, y)的位置上,其中 `getimage()`函数中的参数 `op` 规定如何释放内存中图像。关于这个参数的定义如表 17.5 所示。

表 17.5 putimage()函数中的 op 值

符号常数	数 值	含 义
COPY_PUT	0	复制
XOR_PUT	1	与屏幕图像异或的复制
OR_PUT	2	与屏幕图像或后复制
AND_PUT	3	与屏幕图像与后复制
NOT_PUT	4	复制反像的图形

对于 `imagesize()`函数,只能返回字节数小于 64K 字节的图像区域,否则将会出错,出错时返回-1。

本节介绍的函数在图像动画处理、菜单设计技巧中非常有用。

下面程序模拟两个小球动态碰撞过程。

```
#include<stdio.h>
#include<alloc.h>
#include<graphics.h>          /*引用图形库头文件*/
int main()
{
    int i, driver=DETECT, mode, size;
    void *buf;
    initgraph(&driver, &mode, "");          /*初始化图形模式*/
    setbkcolor(BLUE);                        /*设置背景色*/
    setcolor(LIGHTRED);                     /*设置前景色*/
    setfillstyle(SOLID_FILL, LIGHTGREEN);   /*设置填充样式*/
    circle(100, 200, 30);                   /*画一个圆的母体*/
    floodfill(100, 200, 12);                /*填充圆*/
    size=imagesize(69, 169, 131, 231);      /*计算存储空间*/
    buf=malloc(size);                       /*分配内存*/
    if(!buf) return -1;
    getimage(69, 169, 131, 231,buf);        /*复制里圆*/
    putimage(500, 269, buf, COPY_PUT);      /*粘贴出一个复本*/
    for(i=0; i<185; i++)
```





Note

```

{
    putimage(70+i, 170, buf, COPY_PUT);
    putimage(500-i, 170, buf, COPY_PUT);
    delay(10000);
}
for(i=0;i<185; i++)
{
    putimage(255-i, 170, buf, COPY_PUT);
    putimage(315+i, 170, buf, COPY_PUT);
    delay(1000);
}
getch();
closegraph();
}

```

/*多次粘贴产生两个小球碰撞动画*/

/*设置延时, 缓慢相撞*/

/*多次粘贴产生两个小球碰撞后分开的动画*/

/*延时较小, 快速分开*/

程序运行结果如图 17.9 所示。

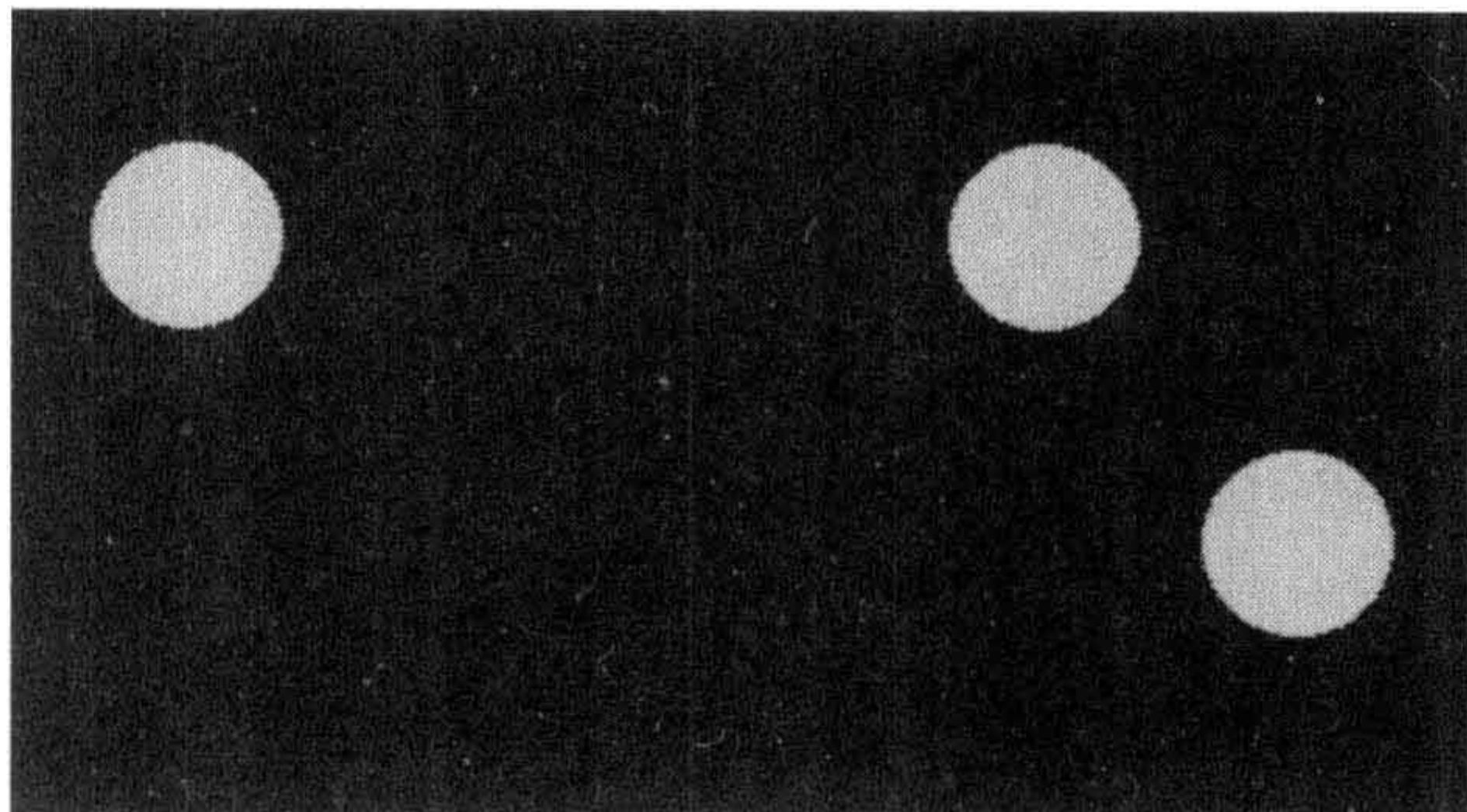


图 17.9 图形复制

专家点评

复制现有图形, 有时比重重新画要省事得多, 因此复制图形功能很有用。

问题 300 怎样在 C 语言中制作动画?

问题阐述

利用 C 语言中的图形函数可以实现动画吗? 怎样实现?

专家解答

动画其实就是快速切换的页面。如果动画中变化的元素比较集中, 可以使用绘画、延时的方法来制作。例如, 在下面的程序中, 先绘制一个逆时针方向逐渐打开的圆, 完全打开后, 第二圈再按逆时针方向逐渐改变成淡颜色, 第三圈擦除图形。



Note

```

#include<stdio.h>
#include<math.h>
#include<graphics.h>                                /*引用图形库头文件*/
main()
{
    int driver,mode;
    int i,maxx,maxy,x,y,r,colormax;
    detectgraph(&driver,&mode);                        /*检测图形驱动程序，图形模式*/
    initgraph(&driver,&mode,"d:\\tc");                /*初始化图形模式*/
    maxx=getmaxx()/2;                                /*获取屏幕中心坐标*/
    maxy=getmaxy()/2;
    r=maxx<maxy?maxx:maxy;                            /*将圆半径设置为屏幕中心的较小值*/
    colormax=getmaxcolor();                          /*最大颜色数*/
    for(i=0;i<360;i++)                                /*用直线画一个辐射的圆*/
    {
        setcolor(i%colormax+1);                    /*每条直线使用不同的颜色*/
        x=maxx+r*sin(i*3.1416/180);                /*计算直线在圆上另一点的坐标*/
        y=maxy+r*cos(i*3.1416/180);
        line(maxx,maxy,x,y);                        /*画线*/
        delay(10000);                               /*短延时*/
    }
    sleep(1);                                         /*长延时*/
    for(i=0;i<360;i++)
    {
        setcolor(i%3+1);                            /*只用蓝、绿、青3种不鲜艳的颜色绘图*/
        x=maxx+r*sin(i*3.1416/180);                /*计算直线在圆上另一点的坐标*/
        y=maxy+r*cos(i*3.1416/180);
        line(maxx,maxy,x,y);                        /*画线*/
        delay(10000);                               /*短延时*/
    }
    sleep(1);                                         /*长延时*/
    setcolor(getbkcolor());                          /*设置背景色为画线颜色*/
    for(i=0;i<360;i++)
    {
        x=maxx+r*sin(i*3.1416/180);
        y=maxy+r*cos(i*3.1416/180);
        line(maxx,maxy,x,y);
        delay(10000);
    }
    closegraph();                                    /*关闭图形模式*/
}

```

程序运行结果如图 17.10 所示。



Note

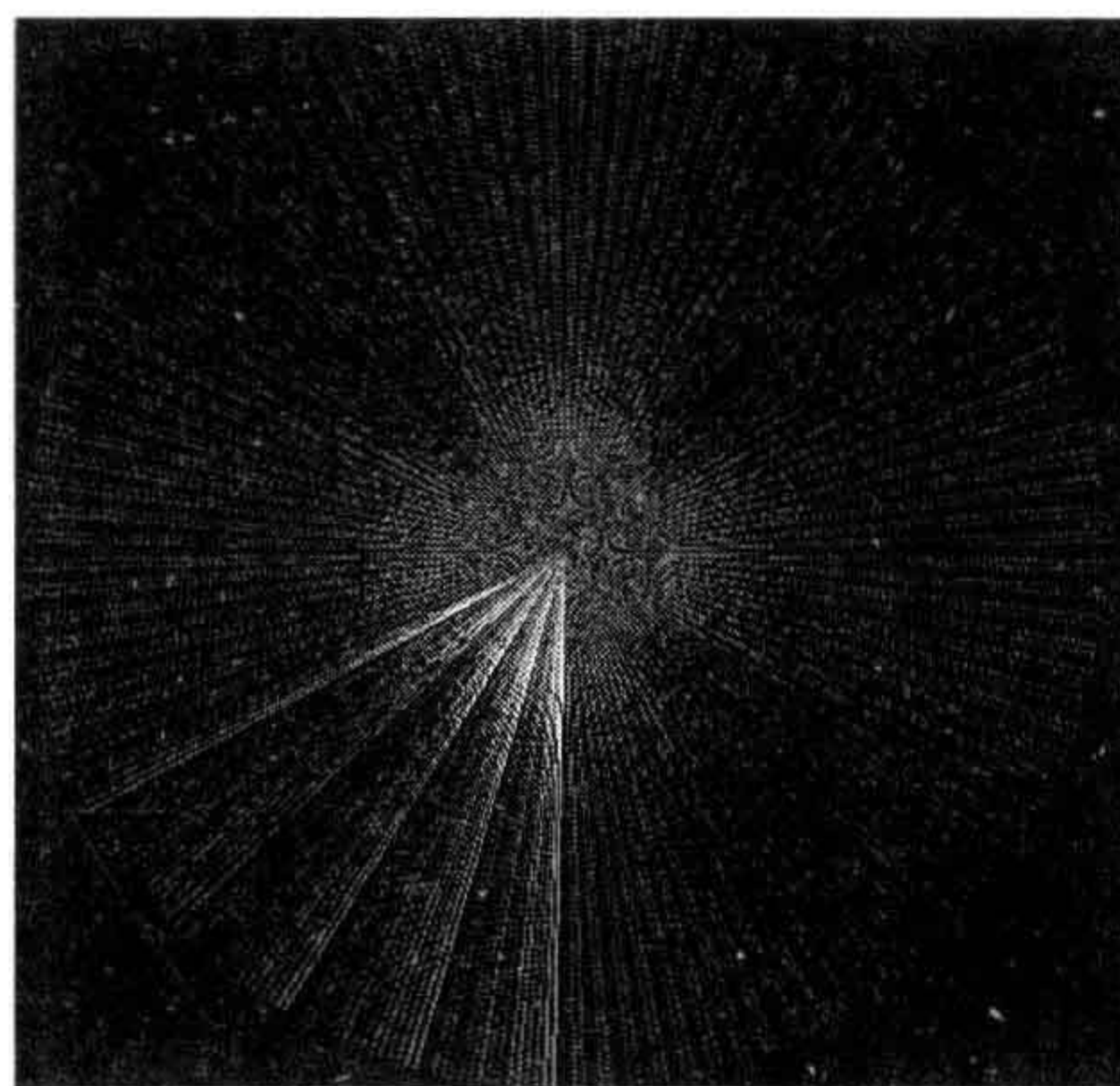


图 17.10 动画的生成

专家点评

动画有很多种形式，上面讲的只是其中一种。只要知道它的基本原理，就可以作出各种形式的动画。